VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA STROJNÍHO INŽENÝRSTVÍ
FACULTY OF MECHANICAL ENGINEERING

ÚSTAV AUTOMATIZACE A INFORMATIKY
INSTITUTE OF AUTOMATION AND COMPUTER SCIENCE

VÝVOJ APLIKACÍ PRO ANDROID

DEVELOPMENT OF ANDROID APPLICATIONS

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE — Bc. Martin Husa
AUTHOR

VEDOUCÍ PRÁCE — Ing. Jan Roupec, Ph.D.
SUPERVISOR

BRNO 2019

ZADÁNÍ VŠKP 1

(tento list nahradíte oficiálním zadáním práce)

ABSTRAKT

Tato diplomová práce zahrnuje rešeršní studii operačního systému Android, principy programování aplikací herního typu a praktický příklad hry pro Android. Hra by se svým žánrem dala zařadit mezi 2D „plošinovky“. Aplikace spolu s textem práce bude sloužit jako pomůcka při výuce programování na VUT FSI nebo jako manuál pro vývoj herních aplikací aplikací.

ABSTRACT

This Master’s thesis contains search study about operating system Android, principles of game application programming and practical example of game for Android. The genre of game can be classify as 2D platformer. Application with thesis text will serve as learning tool in VUT FSI or as manual for game developing.

KLÍČOVÁ SLOVA

Java, Android, Android Studio, Software, Mobilní aplikace, Objektově orientované programování

KEYWORDS

Java, Android, Android Studio, Software, Mobile application, Object oriented programming

BIBLIOGRAFICKÁ CITACE

Bc. HUSA, Martin. *Vývoj aplikací pro Android*, Brno, 2019. Diplomová práce. Vysoké učení technické v Brně, Fakulta strojního inženýrství, Ústav automatizace a informatiky.

PODĚKOVÁNÍ

Chtěl bych poděkovat vedoucímu mojí práce Ing. Janu Roupcovi, Ph.D. za užitečné rady a ochotnou spolupráci.

ČESTNÉ PROHLÁŠENÍ

Prohlašuji, že tato práce je mým původním dílem, zpracoval jsem ji samostatně pod vedením Ing. Jana Roupce, Ph.D. a s použitím literatury uvedené v seznamu literatury.

V Brně dne 1. 3. 2019 ……

Bc. Martin Husa

OBSAH

1 ÚVOD

Smyslem práce je vývoj mobilní hry pro operační systém Android, která, společně s podrobným textem práce, bude sloužit jako manuál pro vývoj podobných aplikací, ale také i jako pomůcka při výuce programování.

Hry se v dnešní době stávají, hlavně díky hardwarové pokročilosti mobilních telefonů, stále populárnější. Her pro operační systém Android, ale i pro jeho největšího konkurenta iOS, každým dnem stále rychleji přibývá. Mobilní hry se svojí komplexností stávají již téměř konkurence-schopnými s PC hrami. Hry na mobilní telefon, mají oproti PC hrám, obrovskou výhodu, v tom, že je hry možné hrát naprosto odkudkoliv a kdekoliv. Lze téměř reálně celkem reálně uvažovat, že herní průmysl se bude v budoucnosti vyvíjet právě tímto směrem.

Práce popisuje základní principy vývoje her, jak už z hlediska programovacího jazyka, tak i z hlediska práce s grafickými texturami. Jako vývojové prostředí byl zvolen program Android Studio od společnosti Google. Do Android Studia byl doinstalován engine pro tvorbu her libGDX a jediný hratelný level byl vytvořen v programu Tiled. Veškeré použité programy pro tvorbu hry jsou zdarma.

Herní aplikace, kterou tato práce popisuje, je naprogramována pro operační systém Android, a to vzhledem k mnohem větší otevřenosti pro vývojáře, než je tomu jiných platforem. Dalším důvodem je, že mobilní telefony s operačním systémem Android jsou jednoznačně nejprodávanější.

Jedná se o 2D hru tzv. „plošinovku“ na motivy 90. let, jejichž herní náplní by se hra dala zařadit do žánrů, jako jsou Arkády, Akční nebo „Skákačky“. Smyslem hry je s jedinou hratelnou postavou, chodící houbou Brunem, projít celou herní mapu, plnou překážek a nepřátel a na konci se potkat s druhou postavičkou Frederikem, čímž úroveň končí. Ve hře se nacházejí objekty, které když Bruno sesbírá, navýší jeho herní score. Veškeré score se sčítá a na konci hry vyhodnotí. Level je možné hrát opakovaně a snažit se o co nejvyšší score.

2 ANDROID

Celá tato kapitola, která popisuje operační systém Android, je kompletně zpracována podle zdroje [1] a [2].

Android (Obr.1) je jeden z mála operačních systémů podporujících mnoho platforem nejrůznějších značek. Z důvodu této otevřenosti má Android filozofii open-source, tzn. jeho zdrojový kód je otevřen vývojářům o mnoho více, než je tomu u konkurence. Díky open-source filozofii má Android velmi rychlý vývoj, ale vznikají zde problémy s optimalizací na konkrétní druh mobilního telefonu.

Obr. 1: Logo operačního systému Android

2.1 Specifikace

Operační systém je postavený na linuxovém jádře, které je specificky upraveno pro využití na mobilních zařízeních s ARM procesory[1].

Android aplikace nekomunikují přímo s jádrem, ale s Android API, které přistupuje k jednotlivým funkcím telefonu, např. fotoaparát, polohový senzor, atd.. O chod aplikace se stará Dalvik VM[2], který funguje tak, že přeloží Java kód, aby byl spustitelný pro mobilní zařízení. Specifikace Androidu je uvedena na obr. 2.

[1] Mikroprocesor s nízkou spotřebou elektrické energie
[2] Virtuální stroj, který vytváří běhové prostředí pro Android aplikace

Konektivita	podpora GSM, CDMA, UMTS, LTE, IDEN, EV-DO, Bluetooth, Wi-Fi, WiMAX, Bluetooth
Aplikace	Dalvik Virtual Machine, instalace bez omezení, multitasking
Data	pro ukládání dat je využívána databáze SQLite
Multimedia	standardně podporuje Android (v závislosti na verzi) formáty WebM, H.263, H.264, MPEG-4 SP, AMR, AMR-WB, AAC, MP3, MIDI, Ogg Vorbis, FLAC, WAV, JPEG, PNG, GIF, BMP a streamované formáty RTP/RTSP, HTML (HTML5 <video> tag), Adobe Flash (RTMP), Apple HTTP Live Streaming
Hardware	Android obsahuje podporu pro dotykový displej (multitouch), GPS, kompas, proximity senzor, gyroskop, akcelerometr, tlakový senzor, teploměr, grafický akcelerátor (2D, 3D), fotoaparát, grafické čipy
Lokalizace	aktuálně podpora více než 50 jazykových verzí a postupně je překládán do dalších
Web prohlížeč	vestavěný webový prohlížeč používá jádro webkit a javascriptový engine V8, tedy stejný jako v prohlížeči Google Chrome

Obr. 2: Specifikace Androidu

2.2 Trh

Mobilní zařízení s Androidem se složitě odlišují od konkurence, prodávající stejný operační systém. Důvodem je, že se jednolivé firmy podílejí na vývoji jen formálně. Zákazníci často hledají „telefon s Androidem“ a na identitě výrobce jim už nezáleží tolik jako dřív. Mezi nejvýznamnějš firmy prodávající mobilní zařízení s Androidem patří např. Samsung, HUAWEI, Xiaomi, atd.

V počtu prodaných kusů Android překonal dosud vedoucí Symbian[3] již v roce 2010. Každý den je s tímto operačním systémem aktivováno přibližně 350 tisíc nových přístrojů, s celkovým zastoupením 130 milionů.

Android svým vstupem na trh významně zamíchal kartami. Dokázal zaujmout výrobce, ale i především uživatele a postupem času se výrazně prosadil na trhu. Srovnání Androidu s konkurencí je uvedeno na obr. 3.

[3] Platforma navržená do mobilních telefonu značky Nokia.

	Android	Windows Phone 7	Symbian	iOS
Zdrojové kódy	OpenSource	Proprietární	Uzavřené	Proprietární
Počet aplikací	200 000+	13 000+	40 000+	330 000+
Multitasking	ano	omezený	ano	omezený
Nových telefonů denně	350 000	31 000	350 000	300 000
Nejúspěšnější výrobci	HTC, Samsung, Motorola	HTC, Samsung	Nokia	Apple
Vývoj aplikací	Java (API)	C# (XNA framework)	C++, QT	Objective-C (Cocoa framework)
Poplatek z každé prodané aplikace	30%			
Registrace vývojáře	Jednorázově 25$	99$/rok	1€ jednorázově	99$/rok

Obr. 3: Srovnání Androidu s konkurencí

3 VÝVOJOVÁ PROSTŘEDÍ

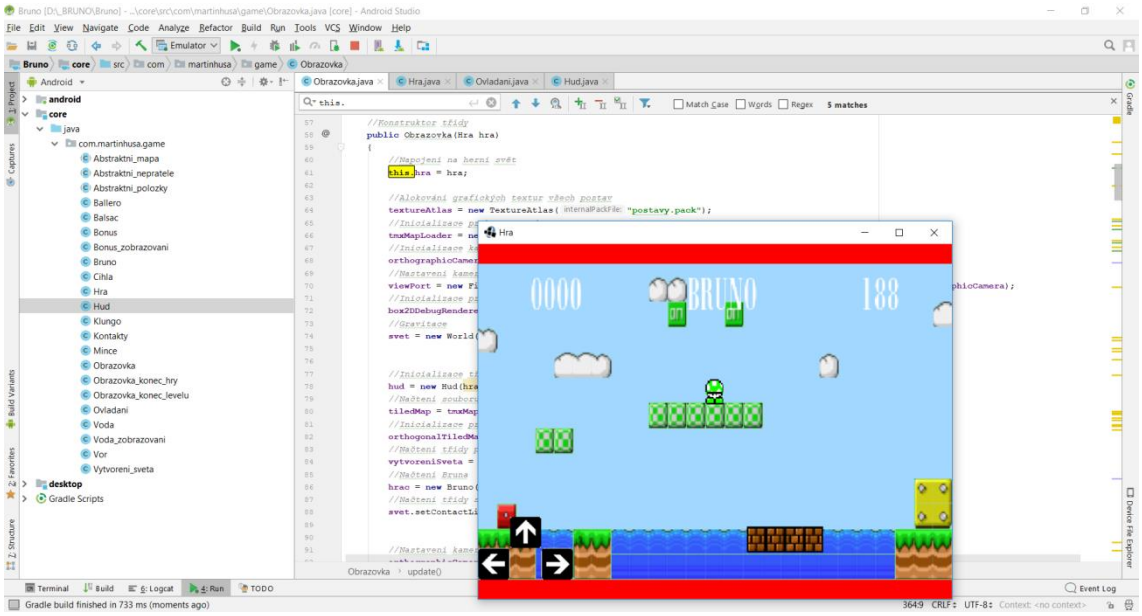
Android aplikace lze vyvíjet v mnoha vývojových prostředích např. Eclipse, NetBeans, Intellij IDEA a Android Studio. V dnešní době je doporučených vývojovým prostředím Android Studio[5]. Dle mých osobních zkušeností je Android Studio mnohem svižnější, vykazuje menší chybovost, práce je v něm obecně mnohem pohodlnější a i instalace je jednoduší.

3.1 Android Studio

Android Studio (Obr.4) je vývojové prostředí pro operační systém Android vytvořené společností Google. Program je založený na IntelliJ IDEA a funguje na všech hlavních platformách. Android Studio má velkou podporu, jak už od vývojářů na oficiálních stránkách: https://developer.android.com/studio, tak i od obrovské komunity vývojářů.

Obr. 4: Logo vývojového prostředí Android Studio[6]

Veškeré aplikace v Android Studiu se píší v programovacím jazyku Java, jehož editaci usnaďňuje inteligentní našeptávač, který značně zrychluje psaní kódu. Je zde obsažen i simulátor reálného zařízení, tvz. emulátor (Obr. 5), na kterém lze funkčnost aplikace okamžitě otestovat.



Obr. 5: Ukázka emulátoru

3.2 libGDX

Celá tato kapitola, popisující libGDX engine, je kompletně zpracována podle zdrojů [3] a [4].

V multifunkční knihovně libGDX, je možné vyvíjet hry, jak pro klasické platformy jako jsou PC nebo Mac, tak i pro mobilní telefony.

Jedná se o herní knihovnu fungující na mnoha platformách. Je založena na OpenGL, jenž spadá pod licenci firmy Apache a je tedy open-source. Platformní kompatibilita je opravdu široká a zahrnuje např. Windows, Linux, Mac OS X, Android, BlackBerry, iOS, Java Applet a Javascript/WebGL (Chrome, Safari, Opera, Firefor, IE přes Google Chrome Frame). Engine prakticky funguje jako překladač do dalších platforem jako je např. Android.

Velká výhoda libGDX je nenáročnost na výkon hardwarového zařízení, protože vyvíjené aplikace je možné spouštět jako Java aplikaci. Je zde možné samozřejmě přehrávat hudbu a to i ve více formátech, obsahuje podporu pro myš, klávesnici a dotykovou obrazovku. Pro vývoj her je možné využít i podporu akcelorometru, kompasu a naprogramovatelnost jednoduchých gest. Z fyzikálního a matematického hladiska engine podporuje vektory, matice i kvaterniony a zahrnuje i geometrické tělesa, jako jsou např. kruhy, obdelníky, atd.

Engine libGDX (Obr.6) funguje jako plugin, který je možné doinstalovat do dvou vývojových prostředí: Eclipse a Android Studio. U Eclipse se jedná spíše o historii, v současnosti je doporučené vývojové prostředí Android Studio.

Obr. 6: Logo libGdx [4]

3.2.1 Box2D

Součástí libGDX je engine Box2D, který pracuje jako dvoudimenzonální fyzikální simulátor. Pro všechy objekty tedy platí fyzikální zákony, které engine umožňuje editovat. Objekty v Box2D se nazývají Body a mají své atributy tzv. Fixture.

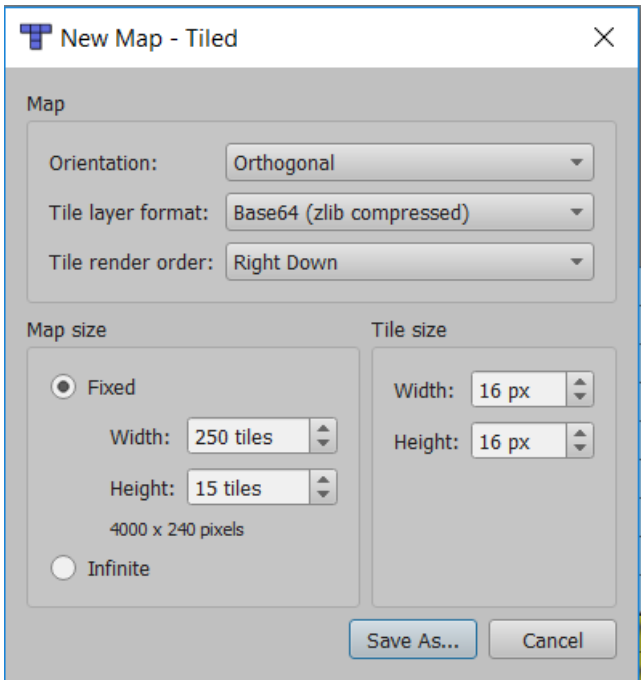
V editoru existují tři typy Body. DynamicBody se používá pro hratelnou postavu. Je ovlivněno gravitací a definují se mu rychlosti pohybu. StaticBody je statický typ Body, nelze tedy jeho polohu ovlivnit v průběhu hry. KinematicBody jsou ovlivněny pouze gravitací nikoliv pohybem ostatních Body.

3.3 Tiled

Program Tiled je ortogonální mapový editor, který není součástí Android Studia, ale funguje jako samostatný program. Program Tiled je zcela zdarma a jeho provoz je zajištěn pomocí doprovolného přispívání. Tiled byl v této práci použit pro vytvoření herní mapy.

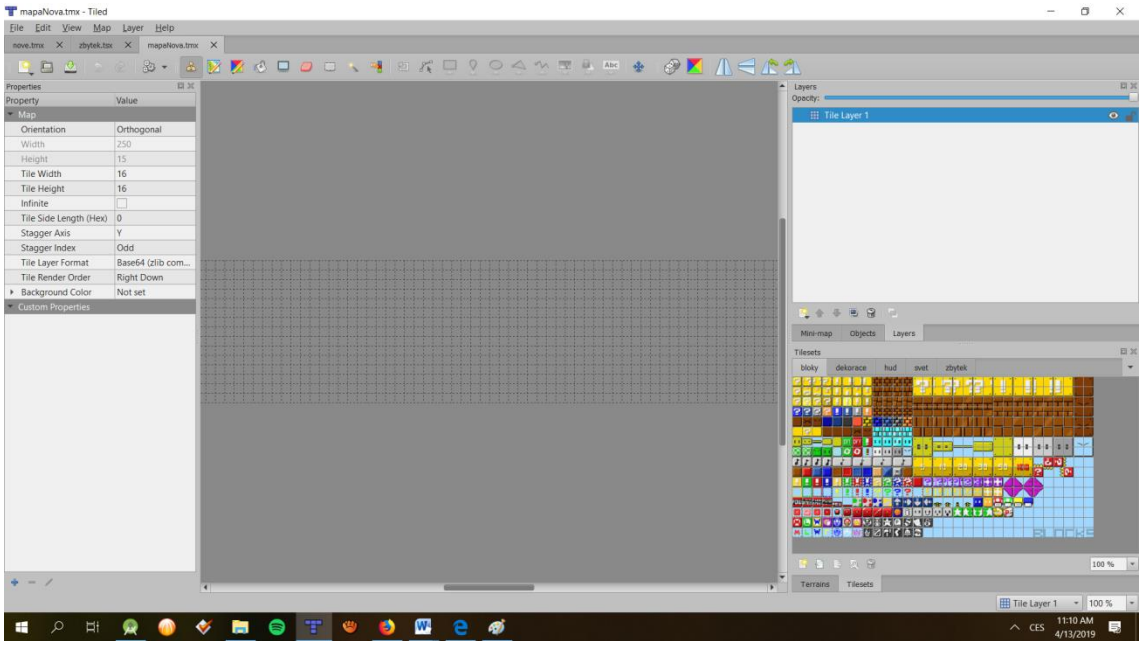
Veškeré grafické textury, použité ve hře, byly staženy z www.spriters-resource.com a naimplementovány do Tiled.

Při tvorbě nového projektu (Obr.7) se nejprve určí velikost hracího pole a ostatní parametry, např. kolmá orientace textur. Pro potřeby této hry je nastavena velikost jednotlivých textur na 16*16 pixelů a level tvoří 250 těchto dílků na délku a 15 na výšku.



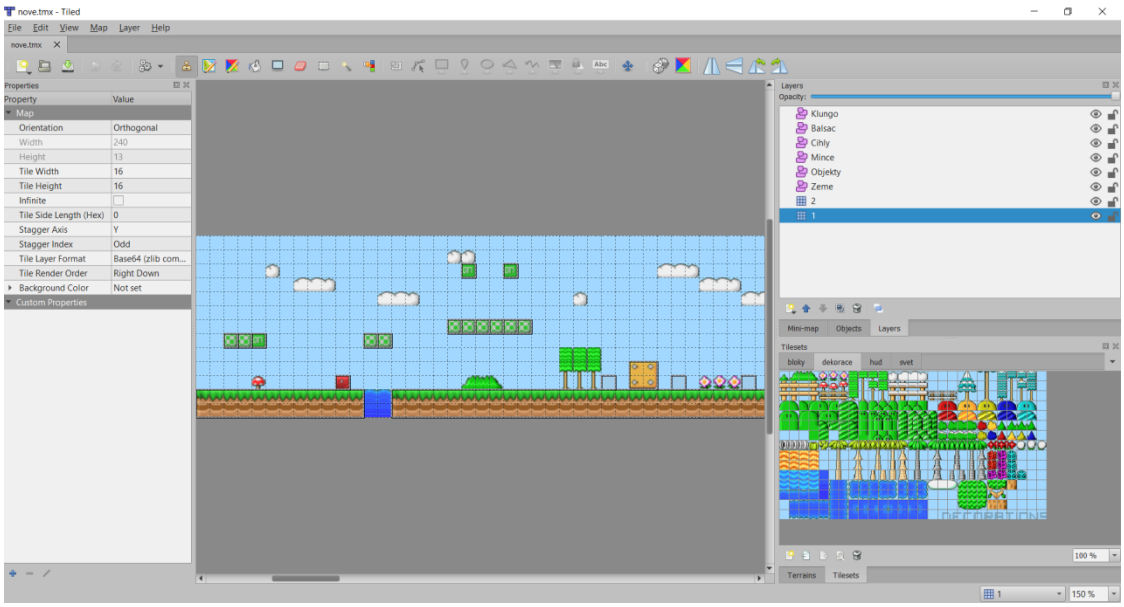
Obr. 7: Zakládání nového projektu

Program tedy vytvoří pravoúhlou strukturu podle zadaných rozměrů (Obr.8), a následně se do obrazovky Tileset, v pravém dolním rohu, naimplementují grafické textury.



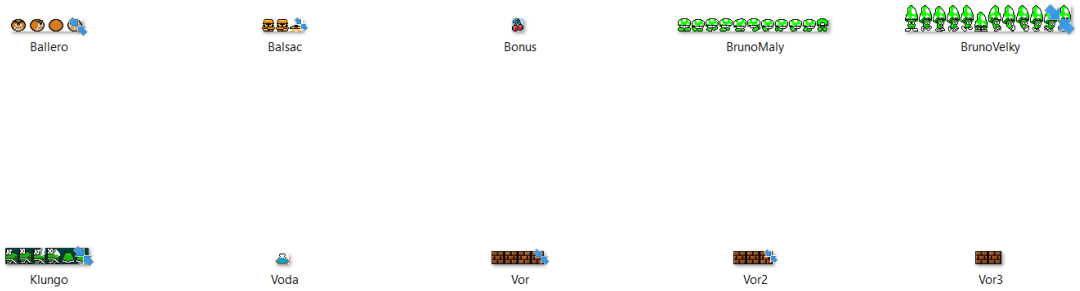
Obr. 8: Pravoúhlá struktůra projektu

Poté je potřeba definovat vrstvy tzv. Layers (Obr.9). Pro pozadí a grafiku jsou použity vrstvy typu Tile, do kterých se celý level manuálně nakreslí. Následně jsou vytvořeny specifické vrstvy typu Object, ke kterým se následně přiřadí objekty, které ve hře budou přicházet do kontaktů. Objekty zahrnují i veškeré nepřátele.



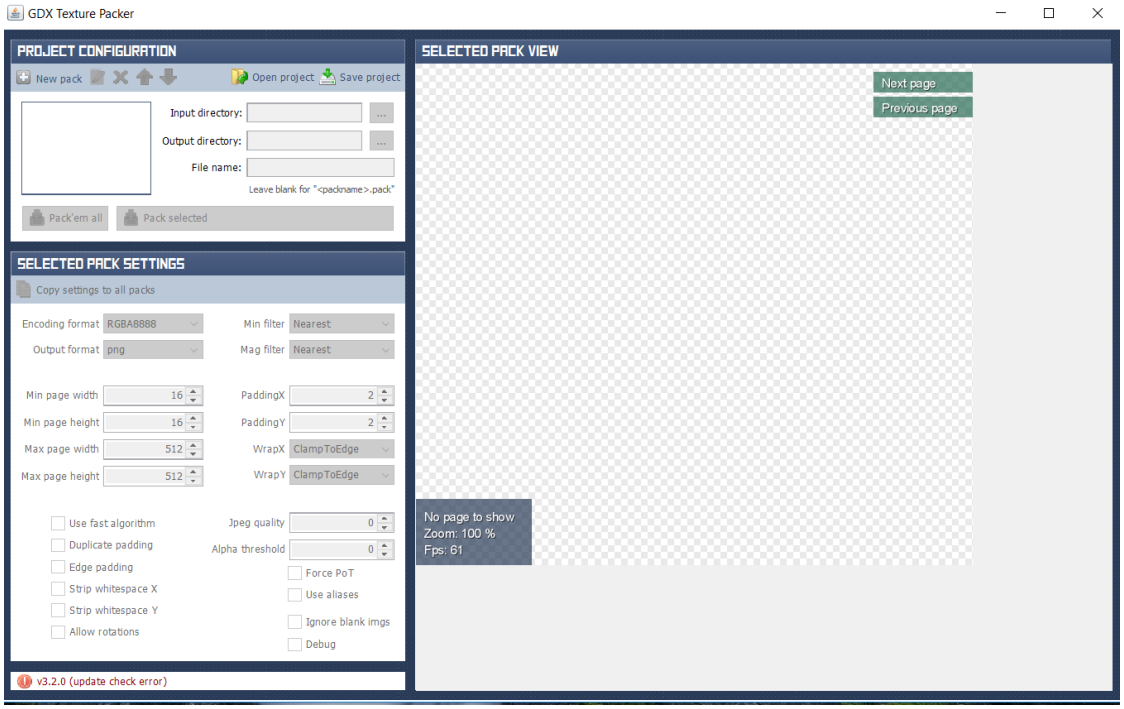
Obr. 9: Vykreslení levelu

Grafické textury pro animaci Bruna a ostatních nepřátel (Obr.10) byly také staženy ze stránky www.spriters-resource.com. Jelikož postavy vykonávají různé pohyby, je jejich animace složena z více textur. Je třeba brát v potaz, že animace musí být ukládány ve formátu png s aktivním parametrem: průsvitnost, aby se za texturou vykreslilo pozadí.



Obr. 10: Animace textur

Pro jednodušší zacházení s texturami byly jednotlivé grafické animace sloučené do jednoho souboru, k tomu byl použit samostatný program libGDX Texturepacker (Obr.11). Program udělal jednak jednoduchou koláž z animací (Obr.12), ale také zakódoval dilčí textury, takže je k ním následně možné přistupovat pomocí kódu v z Android Studiu. Následně byl složený soubor alokovám do složky assest v Android Studiu.



Obr. 11: Program LibGDX Texturepacker

Obr. 12: Složené textury animací postav

4 PROGRAM

Program této hry je napsán v Java kódu ve vývojovém prostředí Android Studio, do kterého byl doinstalován engine pro vývoj her, libGDX. Design hrací plochy je vytvořen ve vývojovém prostředí Tiled, pro úpravu grafických textur byl použit program GDX Texture Packer a vytvoření fontu k zobrazování textů bylo provedeno programem Hiero[4].

Jedinou hratelnou postavou hry je chodící houba Bruno. Úkolem Bruna je projít celou mapu, plnou nepřátel a překážek a dostat se až na konec ke svému příteli Frederickovi. Ve hře se vyskytují tři typy nepřátel – trpaslík Balsac, skákající míč Ballero a želva Klungo.

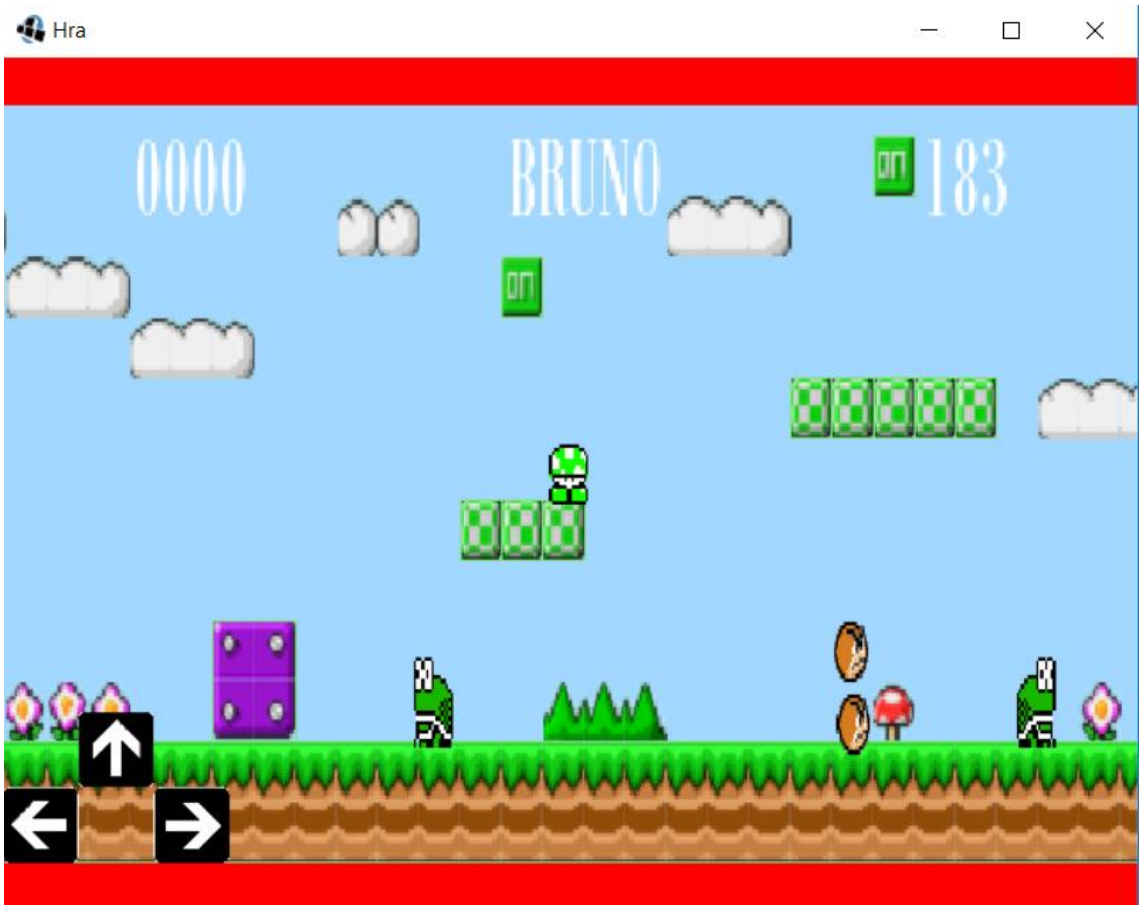
Bruno je schopen vykonávat tři druhy pohybů – vertikálně vpravo i vlevo a horizontálně vyskakovat respektivě seskakovat. Ke zničení Bruna může dojít dvěma způsoby – zabitím od nepřítele nebo vyskočením z mapy.

Společná vlastnost všech typů nepřátel je, že pokud narazí do okolí nebo do sebe navzájem, začnou vykonávat reverzní pohyb. Nemohou se tedy ve výchozím stavu sami navzájem zničit. První a základní typ nepřítele je tzv. Balsac. Balsac je trpaslík, kterého je možné zničit skokem na hlavu. Pokud dojde k jinému kontaktu s Brunem než je skok na Balsacovu hlavu, je zničen Bruno. Další obdobný typ nepřítele je tzv. Ballero, který vykonáva pohyb pouze v ose Y a Bruno ho není schopen sám zničit. Poslední a složitější typ nepřítele je Klungo. Pokud Bruno na Klunga skočí, Klungo změní svůj stav a zabalí se na určitý krátký čas do krunýře. Pokud dojde k jinému kontaktu s Brunem než je skok na Klungovu hlavu, je zničen Bruno. Když je Klungo v krunýří a Bruno ho z nějaké strany kontaktuje, krunýř Klunga začne vykonávat opačný směrem zrychlený pohyb, který může zničit jak ostatní nepřátele, tak i Bruna.

Ve hře se nachází zelené čtvercové boxy s označením ON, se kterými je Bruno schopen pomocí své hlavy interagovat. Při kontaktu hlavy a boxu, box zčervená a objeví se na něm nápis OFF. Současně se zvýší score 100 nebo se objeví bonusové položky: - džbán s vodou nebo třešně. Džbán i třešně poté začnou vykonávat pohyb směrem nahoru. Pokud dojde ke kontaktu džbánu s Brunem, jeho postava se dvojnásobně zvětší. Toto zvětšení znamená, že Bruno má prakticky o jeden život navíc, protože když následně dojde k zabití, jeho velikost se vrátí do původních rozměrů. Zvětšení také zapříčiní, že Bruno může úderem své hlavy ničit další čtvercové boxy tzv. cihly. Zničení každého boxu přičte Brunovi 50 bodů do score. Pokud dojde ke kontaktu s třešněmi, je hráčovo score navýšeno o 500.

V horní části hrací plochy se nachází tzv. HUD menu. Zde je zobrazeny tři typy informací. Score, které hráč během hry získává, logo s názvem hry, a odpočet času, který má hráč na dokončení levelu. V levém dolním rohu se nachází joystick, pro ovládání postavy. Ukázka hry je znázorněna na obr. 13.

[4] Nástroj pro bitmapové písmo, použitelné v aplikacích libGDX.



Obr. 13: Ukázka hry

4.1 Třída Hra

První třída, která v této práci bude popisována, se nazývá Hra. Tato třída je rozšířena o prvek `Game`, který je extrahovaný z pluginu libGDX.

```
public class Hra extends Game
```

V této třídě jsou definovány dvě proměnné typu `int`: **sirka** a **vyska**, definující velikost hrací plochy:

```
//Proměnné rozměrů promítací obrazovky
public static final int sirka = 500;
public static final int vyska = 210;
```

Do proměnné `float` **korekce**, je nastavená hodnota 100. Proměnná znázorňuje hodnotu 100 pixelů na jeden metr a používá se pro korekci hodnot všech rozměrů ve hře. Důvodem je nastavení jednotek v enginu Box2D.

```
//Pixelová korekce
public static final float korekce = 100;
```

Jsou zde definovány proměnné typu `short` pro bity, přiřazené k jednotlivým objektům, které jsou následně používány v kontaktní logice.

```
//Proměnné pro filtry fixtur
public static final short prazdny_bit = 0;
public static final short zeme_bit = 1;
public static final short bruno_bit = 2;
public static final short cihla_bit = 4;
public static final short mince_bit = 8;
public static final short zniceny_bit = 16;
public static final short objekty_bit = 32;
public static final short nepratele_bit = 64;
public static final short hlava_nepritele_bit = 128;
public static final short vec_bit = 256;
public static final short hlava_bruna_bit = 512;
public static final short ballero_bit = 1024;
public static final short okraje_bit = 2048;
public static final short vor_bit = 4096;
public static final short vor_hlava_bit = 8192;
public static final short bonus_bit = 16384;
```

Pro potřeby celé hry je z důvodu paměťové náročnosti vytvořen pouze jeden prvek `SpriteBatch`. Tento prvek prakticky slouží jako kontejner pro grafiky, textury atd., které jsou potřeba rendrovat na hrací plochu.

```
public SpriteBatch spriteBatch;
```

Dále je zde vytvořena funkce *create()*. Ve funkci je do proměnné **`spriteBatch`** inicializován prvek `SpriteBatch`. Pomocí funkce *setScreen()*, je inicializována hrací plocha k další třídě s názvem Obrazovka.

Ve hře je použito několi rozdílných zvuků, např. při smrti Bruna, konci hry, atd. Se zvukovými soubory se pracuje následovně, do proměnné `AssetManager` **`assetManger`** jsou pomocí funkce *load()* alokovány audio soubory. Parametr `Music.class`, resp. `Sound.class` specifikuje druh souboru. V poslední řadě je funkcí *finishLoading()*, zavolané do **`assetManger`**, ukončeno načítání audio souborů.

```
//Funkce pro vytvoření hudby, atd.
@Override
public void create()
{
   spriteBatch = new SpriteBatch();

    //Nastavení assetmanageru
   assetManager = new AssetManager();
   assetManager.load("audio/hudba/hudba1.mp3", Music.class);
   assetManager.load("audio/zvuky/mince.wav", Sound.class);
   assetManager.load("audio/zvuky/neznicenacihla.wav", Sound.class);
   assetManager.load("audio/zvuky/znicenacihla.wav", Sound.class);
   assetManager.load("audio/zvuky/predmet.wav", Sound.class);
   assetManager.load("audio/zvuky/zvetseni.wav", Sound.class);
   assetManager.load("audio/zvuky/zmenseni.wav", Sound.class);
   assetManager.load("audio/zvuky/zabiti.wav", Sound.class);
   assetManager.load("audio/zvuky/smrt.wav", Sound.class);

   assetManager.finishLoading();

   //Napojení na herní svět
       setScreen(new Obrazovka(this));
}
```

Třída obsahuje také funkci *dispose()*, která slouží pro uzavírání prvků z důvodu úspory operační paměti. Jsou zde uzavřeny audio soubory v proměnné **`assetManager`**.a prvek `spriteBatch`, v proměnné **`spriteBatch`**.

```
//Funkce pro zavírání prvků
@Override
public void dispose()
{
   super.dispose();
   assetManager.dispose();
   spriteBatch.dispose();
}
```

4.2 Obrazovka

Tato třída slouží pro vykreslení herní mapy. Nastavuje se zde to, co má kamera pro hráče vykreslovat, jakou hudbu za jakých situací má program spustit a jsou zde definované uživatelské vstupy neboli ovládaní hry.

Do třídy Obrazovka je naiplementován prvek `Screen` z pluginy libGDX, pomocí kterého jsou do třídy naimplementovány funkce, běžně používané pro tento typ třídy.

```
public class Obrazovka implements Screen
```

Je zde vytvořen konstruktor: *Obrazovka()*. Konstruktor pracuje s již dříve nadefinovanou proměnou třídy `Hra` **`hra`**. Nejdůležitější části konstruktoru je funkce *OrthographicCamera()*, zavolaná do proměnné `OrthographicCamera` **`ortographicCamera.`** `OrthographicCamera` promítá to, co zobrazuje funkce *FitViewport()*, zavolaná do proměnné `Viewport` **`viewport.`** `FitViewport` je typ gameportu, který zachovává poměr stran i přes měnící se velikost obrazovky. Následně je ještě kamera vycenterována do středu hrací plochy podělením rozměrů hrací plochy dvěma.

V konstruktoru dochází k propojení programu s grafikou, vytvořenou v mapovém editoru Tiled. Pomocí proměnných a funkcí: `TmxMapLoader` **`tmxMapLoader`** a *TmxMapLoader()*, `TiledMap` **`tiledMap`** a *load()*, `OrthogonalTiledMapRenderer` **`orthogonalTiledMapRenderer`** a *OrthogonalTiledMapRenderer()*.

Soubor s grafickým texturami k postavám je alokován pomocí proměnné `TextureAtlas` **`textureAtlas`** a funkce *TextureAtlas()*.

Do proměnné `World` **`svet`** je zavolána funkce *World()* s parametrem *Vector2()*, který definuje gravitaci. Parametr `true` slouží pro uspávání objektů v nečinnosti, z důvodu úspory paměti.

Do proměnné `World` **`svet`** je ještě zavolána funkce *setContactListener()* s parametrem třídy *Konatkty()*, která obsahuje komplentní kontaktní logiku mezi objekty.

Pomocí proměnné `Music` **`hudba`** a funkce *get()* je načten audio soubor z proměnné assetManager ze třídy `Hra`. Funkcí *setLooping()* s atributem `true` je zadefinováno cyklické přehrávání souboru a funkcí *play()* je pak audio soubor následně spuštěn.

Pro zobrazování předmětů jsou zde vytvořeny proměnné `Array<Vec>()` **`vecArray`** a `LinkedBlockingQueue<VecDef>` **`vecZobrazeni`**, které jsou použity v dalších funkcích v této třídě.

```
//Konstruktor třídy
public Obrazovka(Hra hra)
{
    //Napojení na herní svět
    this.hra = hra;

    //Alokování grafických textur všech postav
    textureAtlas = new TextureAtlas("postavy.pack");
    //Inicializace prvku TmxLoader
    tmxMapLoader = new TmxMapLoader();
    //Inicializace kamery
    orthographicCamera = new OrthographicCamera();
    //Nastavení kamery na rozměrové veličiny
    viewPort = new FitViewport(Hra.sirka / Hra.korekce, Hra.vyska /
Hra.korekce, orthographicCamera);
    //Inicializace prvku Box2DDebugRenderer
    box2DDebugRenderer = new Box2DDebugRenderer();
    //Gravitace
    world = new World(new Vector2(0, -10), true);


    //Inicializace třídy HUD
    hud = new Hud(hra.spriteBatch);
    //Načtení souboru s mapou levelu, vytvořeného v programu Tiled
    tiledMap = tmxMapLoader.load("nove.tmx");
    //Inicializace prvku OrthogonalTiledMapRenderer
    orthogonalTiledMapRenderer = new OrthogonalTiledMapRenderer(tiledMap, 1 /
Hra.korekce);
    //Načtení třídy pro vytvoření světa
    vytvoreniSveta = new Vytvoreni_sveta(this);
    //Načtení Bruna
    bruno = new Bruno(this);
    //Načtení třídy s kontaktní logikou
    svet.setContactListener(new Kontakty());


    //Nastavení kamery do středu
    orthographicCamera.position.set(viewPort.getWorldWidth() / 2,
viewPort.getWorldHeight() / 2, 0);


    //Spuštění zvukového souboru
    hudba = Hra.assetManager.get("audio/hudba/hudba1.mp3", Music.class);
    //Opakované přehrávaní zvukového souboru
    hudba.setLooping(true);
    hudba.play();

    //Zobrazovaní položek - voda a bonus
    vodaArray = new Array<Abstraktni_polozky>();
    vodaZobrazovani = new LinkedBlockingQueue<Voda_zobrazovani>();
    bonusArray = new Array<Abstraktni_polozky>();
    bonusZobrazovani = new LinkedBlockingQueue<Bonus_zobrazovani>();

    //Inicializace ovládání - joystick
    ovladani = new Ovladani();
}
```

Funkce *vytvor_voda()*, pomocí funce *add()* s parametrem `Voda_zobrazovani` **voda_zobrazovani**, přidá do proměnné `LinkedBlockingQueue<Voda_zobrazovani>` **vodaZobrazovani** láhev vody, který se má v herním světě zobrazit.

```
//Funkce pro zobrazování - voda
public void vytvor_voda(Voda_zobrazovani voda_zobrazovani)
{
    this.vodaZobrazovani.add(voda_zobrazovani);
}
```

Funkce *vytvor_voda_2()* podmínkou s funkcí *isEmpty()* zjistí, jestli proměnná `LinkedBlockingQueue<VecDef>` **vodaZobrazovani** není prázdná a do proměnné `Voda_zobrazovani` **voda_zobrazovani** se pomocí funkce *poll()* načte hodnota **vodaZobrazovani**. V druhé podmínce se ověří jestli je třída, načtená do **voda_zobrazovani**, rovna třídě Voda. Jestli ano, je do proměnné `Array<Abstraktni_polozky>` **vodaArray** pomocí funkce *add()* vytvořena třída Voda.

```
//Funkce pro zobrazování - voda
public void vytvor_voda_2()
{
    if(!vodaZobrazovani.isEmpty())
    {
        Voda_zobrazovani voda_zobrazovani =   this.vodaZobrazovani.poll();

        //Vytvoří položku podle třídy Voda
        if(voda_zobrazovani.typ == Voda.class)
        {
            vodaArray.add(new Voda(this, voda_zobrazovani.pozice.x,
voda_zobrazovani.pozice.y));
        }
    }
}
```

Funkce pro vytvoření druhé položky ze hry: Bonus, jsou obdobné funkcím pro vytvoření vody.

```
//Funkce pro zobrazování - bonus
public void vytvor_bonus(Bonus_zobrazovani bonus_zobrazovani)
{
    this.bonusZobrazovani.add(bonus_zobrazovani);
}

//Funkce pro zobrazování - bonus
public void vytvor_bonus_2()
{
    if(!bonusZobrazovani.isEmpty())
    {
        Bonus_zobrazovani bonus_zobrazovani = this.bonusZobrazovani.poll();

        //Vytvoří položku podle třídy Bonus
        if(bonus_zobrazovani.typ == Bonus.class)
        {
            bonusArray.add(new Bonus(this, bonus_zobrazovani.pozice.x,
bonus_zobrazovani.pozice.y));
        }
    }
}
```

Interakci s uživatelem zajišťuje třída *uzivatelsky_vstup*. Zde dochází prvně ke kontrole aktuálního stavu Bruna, porovnáním stavu: `Bruno` **`bruno.brunoSoucasnyStav`** a statusu smrti: **`brunoStav.smrt`**.

Je zde nadefinovaný výskok Bruna a obdobně horizontální pohyby vpravo a vlevo. Pomocí funkce *isKeyJustPressed(),* zavolané do `Gdx.input` se zjistí jaká klávesa byla stlačena a funkcí *applyLinearImpulse()*, zavolané do proměnné `Bruno` **`bruno.brunoBody`**, se udělí Brunovi vektor pohybu, ve kterém je nastavena i intenzita pohybu. Funkce *getWorldCenter()*, zavolaná do **`bruno.brunoBody`**, zajistí, aby byl pohybový impulz udělen do středu Brunova těla a poslední parametr true znamená vzbuzení objektu, tím se zajišťuje úspora operační paměti.

Nacházejí se zde dvě vnořené podmínky, které pomocí funkce *getX()*, zavolané do proměnné **`bruno`**, definují oblast levelu, pro kterou jsou upravené uživatelké vstupy. Pro ledový level je zvojnásobena intenzita pohybového impulzu při pohybu doprava a doleva, což zapříčiní dojem, že Bruno po mapě klouže. Pro lítací level je pro skok podmínka, že se Bruno smí odrazit, pouze když je jeho rychlost v ose Y téměř nulová, Bruno tak může prakticky lítat.

```
//Funkce pro definování uživatelských vstupů
public void uzivatelsky_vstup()
{
  //Uživatelský vstup funguje, jen když je Bruno živý, u každého pohybu je
vektor aplikován do středu Body
  if(bruno.brunoSoucasnyStav != brunoStav.smrt)
  {
    //Bruno může vyskočit, jen když je jeho rychlost v y zanadbatelně nulová
    if(Gdx.input.isKeyJustPressed(Input.Keys.UP) &&
bruno.brunoBody.getLinearVelocity().y >= 0 &&
bruno.brunoBody.getLinearVelocity().y < 0.1)
        bruno.brunoBody.applyLinearImpulse(new Vector2(0,4f),
bruno.brunoBody.getWorldCenter(), true);

    //Pohyb vpravo
    if(Gdx.input.isKeyPressed(Keys.RIGHT))
        bruno.brunoBody.applyLinearImpulse(new Vector2(0.1f, 0),
bruno.brunoBody.getWorldCenter(), true);

    //Pohyb vlevo
    if(Gdx.input.isKeyPressed(Keys.LEFT))
        bruno.brunoBody.applyLinearImpulse(new Vector2(-0.1f, 0),
bruno.brunoBody.getWorldCenter(), true);

      //Ledový level - pohyb v X je dvojnásobný
      if(bruno.getX() > 1152 / Hra.korekce && bruno.getX() < 1760 /
Hra.korekce)
      {
          if(Gdx.input.isKeyJustPressed(Input.Keys.UP) &&
bruno.brunoBody.getLinearVelocity().y >= 0 &&
bruno.brunoBody.getLinearVelocity().y < 0.1)
              bruno.brunoBody.applyLinearImpulse(new Vector2(0,4f),
bruno.brunoBody.getWorldCenter(), true);

          if(Gdx.input.isKeyPressed(Keys.RIGHT))
              bruno.brunoBody.applyLinearImpulse(new Vector2(0.2f, 0),
bruno.brunoBody.getWorldCenter(), true);

          if(Gdx.input.isKeyPressed(Keys.LEFT))
              bruno.brunoBody.applyLinearImpulse(new Vector2(-0.2f, 0),
bruno.brunoBody.getWorldCenter(), true);
      }

      //Lítající level - v Y může Bruno skákat, aji když má nenulovou rychlost
      if(bruno.getX() > 2256 / Hra.korekce && bruno.getX() < 3040 /
Hra.korekce)
      {
          //vyskočení v Y ose 4 za sekundu, vyskok aplikovat ve stredu objektu
aby vyskocil rovne a nerocil se, zbudit = true
          if(Gdx.input.isKeyJustPressed(Input.Keys.UP))
              bruno.brunoBody.applyLinearImpulse(new Vector2(0,2.5f),
bruno.brunoBody.getWorldCenter(), true);

          //pohyb vpravo
          if(Gdx.input.isKeyPressed(Keys.RIGHT))
              bruno.brunoBody.applyLinearImpulse(new Vector2(0.1f, 0),
bruno.brunoBody.getWorldCenter(), true);

          //pohyb vlevo
          if(Gdx.input.isKeyPressed(Keys.LEFT))
              bruno.brunoBody.applyLinearImpulse(new Vector2(-0.1f, 0),
bruno.brunoBody.getWorldCenter(), true);
      }
  }
}
```

Funkce *uzivatelsky_vstup_joystick()* definuje uživatelský vstup pro mobilní zařízení, tzv. joystick, který se nachází v levém dolním rohu. Joystick je deklarovaný ve vlastní třídě Ovladani. Funkce je obdobná funkci *uzivatelsky_vstup()*, s těmi rozdíly, že v podmínce pro každý směr pohybu, je volána boolean funkce, ze třídy ovládání, která signalizuje, že joystick byl uživatelem stlačen.

```
//Funkce pro definování uživatelských vstupů od joysticku
public void uzivatelsky_vstup_joystick()
{
    //Uživatelský vstup funguje, jen když je Bruno živý, u každého pohybu je
vektor aplikován do středu Body
    if(bruno.brunoSoucasnyStav != brunoStav.smrt)
    {

        if (ovladani.isSkok() && bruno.brunoBody.getLinearVelocity().y >= 0 &&
bruno.brunoBody.getLinearVelocity().y < 0.1)
        {
            bruno.brunoBody.applyLinearImpulse(new Vector2(0,4f),
bruno.brunoBody.getWorldCenter(), true);
        }

        else if (ovladani.isDoprava())
        {
            bruno.brunoBody.applyLinearImpulse(new Vector2(0.1f, 0),
bruno.brunoBody.getWorldCenter(), true);
        }

        else if (ovladani.isDoleva())
        {
            bruno.brunoBody.applyLinearImpulse(new Vector2(-0.1f, 0),
bruno.brunoBody.getWorldCenter(), true);
        }
    }
}
```

Hra může skončit přesně čtyřmi situacemi: Bruno je zabit nepřítelem, vyskočí z mapy, nestihne level dokončit do 200s a nebo level úspěšně dokončí. Pro zohlednění těchto situací jsou vytvořeny čtyři `boolean` funkce. Funkce *konec_hry_smrt()* porovnává současný stav Bruna se stavem smrti, funkce *konec_hry_dokonceni()*, alokuje místo na konci levelu, kde stojí Frederick, funkce *konec_hry_spadnuti()*, deklaruje, že Brunova poloha v ose Y nesmí být menší než 0 a *funkce konec_hry_cas()*, zavolání funkce *ziskejStavovyCas()*, do proměnné `Bruno` **`bruno`**, ukončí hru po 200s.

```
//Funkce pro skončení hry - zabití Bruna
public boolean konec_hry_smrt()
{
    //Po smrti počká ještě 3s, aby se stihla promítnou animace
    if(bruno.brunoSoucasnyStav == brunoStav.smrt && bruno.ziskejStavovyCas() >
3)
    {
        return true;
    }
    return false;
}

//Funkce pro skončení hry - dokončení levelu
public boolean konec_hry_dokonceni()
{
    //Alokováno místo na konci levelu, kam má bruno dojít
    if(bruno.getX() > 3744 / Hra.korekce && bruno.getX() < 3760 / Hra.korekce
&& bruno.getY() < 50 / Hra.korekce)
    {
        return true;
    }
    return false;
}

//Funkce pro skončení hry - vypadnutí z mapy
public boolean konec_hry_spadnuti()
{
    if(hr bruno ac.getY() < 0 / Hra.korekce && bruno.ziskejStavovyCas() > 1)
    {
        //Zastavení zvukového souboru
        Hra.assetManager.get("audio/hudba/hudba1.mp3", Music.class).stop();
        //Spuštění zvukového souboru
        Hra.assetManager.get("audio/zvuky/smrt.wav", Sound.class).play();

        return true;
    }
    return false;
}

//Funkce pro skončení hry - 200s
public boolean konec_hry_cas()
{
    if(bruno.ziskejStavovyCas() > 200)
    {
        //Zastavení zvukového souboru
        Hra.assetManager.get("audio/hudba/hudba1.mp3", Music.class).stop();
        //Spuštění zvukového souboru
        Hra.assetManager.get("audio/zvuky/smrt.wav", Sound.class).play();

        return true;
    }
    return false;
}
```

Další použitá třída, naimplementovaná z prvku `Screen` je *resize()*, která aktualizuje rozměry `viewPort` a `ovladani` na rozměry hrací plochy.

```
//Funkce pro rozměrový update kamery
@Override
public void resize(int sirka, int vyska)
{
    //Update viewportu
    viewPort.update(sirka, vyska);
    ovladani.resize(sirka, vyska);
}
```

Další funkce, naimplementovaná z prvku `Screen`, sloužící k vykreslování prvků, je *render(),* funkce pracuje s časovou proměnnou `float` **`cas`**. Proměnná cas je pomocí funkce *update()* časově aktualizována. Ve třídě je potřeba prvně pomocí funkcí *glClearColor()* a *glClear()* vyčistit obrazovku. Funkcí *render()*, zavolané do proměnné `OrthogonalTiledMapRenderer` **`orthogonalTiledMapRenderer`** je nastaveno rendrování herní mapy. Pro proměnnou `OrthograpicCamera` **`orthographicCamera`**, je nastaveno kombinované promítání a rendrování herního světa a vykreslen prvek `spriteBatch` ze třídy Hra.

Probíhá zde renderování nepřátel a položek ve hře pomocí cyklu for a funkcí *draw()* zavolanou s parametrem `hra.spriteBatch`.

Jsou zde vytvořeny čtyři podmínky, které spouští obrazovku po skončení hry.

```
//Funkce pro rendrování obrazovky
@Override
public void render(float cas)
{
    update(cas);

    //Vyprázdní obrazovku
    Gdx.gl.glClearColor(1, 0, 0, 1);
    Gdx.gl.glClear(GL20.GL_COLOR_BUFFER_BIT);

    //Nastaveni rendrování kamery, kombinované promítání s nepřáteli
    orthogonalTiledMapRenderer.render();
    hra.spriteBatch.setProjectionMatrix(orthographicCamera.combined);
    box2DDebugRenderer.render(svet, orthographicCamera.combined);
    hra.spriteBatch.begin();
    bruno.draw(hra.spriteBatch);

    //Vykreslení položek - voda a bonus
    for(Abstraktni_polozky abstraktniVoda : vodaArray)
        abstraktniVoda.draw(hra.spriteBatch);
    for(Abstraktni_polozky abstraktniVoda : bonusArray)
        abstraktniVoda.draw(hra.spriteBatch);

    //Vykreslení nepřátel
    for(Abstraktni_nepratele abstraktniNepratele :
vytvoreniSveta.zobrazeniNepratel())
        abstraktniNepratele.draw(hra.spriteBatch);

    hra.spriteBatch.end();

    //Vykreslování toho, co vidí kamera
    hra.spriteBatch.setProjectionMatrix(hud.stage.getCamera().combined);

    //Vykreslení prvku stage
    hud.stage.draw();

    ovladani.draw();

    //Při spuštění konec_hry_dokonceni() nastavit obrazovku
    if(konec_hry_dokonceni())
    {
        hra.setScreen(new Obrazovka_konec_levelu(hra));
        dispose();
    }
    //Při spuštění konec_hry_vypadnuti() nastavit obrazovku
    if(konec_hry_vypadnuti())
    {
        hra.setScreen(new Obrazovka_konec_hry(hra));
        dispose();
    }
    //Při spuštění konec_hry_smrt(), nastavit obrazovku
    if(konec_hry_smrt())
    {
        hra.setScreen(new Obrazovka_konec_hry(hra));
        dispose();
    }
    //Při spuštění konec_hry_cas() nastavit obrazovku
    if(konec_hry_cas())
    {
        hra.setScreen(new Obrazovka_konec_hry(hra));
        dispose();
    }
}
```

Třída vytvořená pro aktualizace se nazývá *update()* a probíhají zde všechny updaty ve hře např. uživatelské vstupy, update kamery, nepřátel, zobrazovaných položek. Třída samozřejmě musí pracovat s časovou proměnnou `float` **`cas`**.

Je zde nastavena pomocí funkce *step()*, zavolané do proměnné `World` **`world`**, rychlost kalkulace herního světa a to rychlostí 60 krát za vteřinu.

Probíhá zde update obou uživatelských vstupů. Pomocí funkce *update()*, s parametrem cas, zavolané do proměnných `Bruno` **`bruno`** a `Hud` **`hud`**, se aktualizuje Bruno a Hud menu.

Probíhá zde i update prvku `OrthographicCamera`. Ta musí fungovat tak, aby se posouvala za Brunem v ose X, proto je její parametr **`position.x`** dán do rovnosti s X-ovou souřadnící Bruna, zjištěnou pomocí funkce *getX*.

Pomocí funkce *setView()*, s paramtrem **`orthographicCamera`**, je proměnná `OrthogonalTiledMapRenderer` **`orthogonalTiledMapRenderer`** nastavena pouze na rendrování toho, co vidí kamera.

Pro update nepřátel je vytvořena podmínka for, ve které probíhá update opět pomocí funkce *update()* s parametrem **`cas`**, zavolané do třídy Abstraktni_nepratele. Všichni nepřátelé se aktivují v závislosti na poloze Bruna. Pokud není Bruno blíž jak 200 / **`Hra.korekce`**, nepřátelé jsou nečinní, čímž se šetří operační paměť. Tato logika je vytvořena pomocí podmínky, ve které se funkcí *getX()*, zavolané do proměnné **`bruno`**, zjistí x-ová poloha nepřítele a Bruna. Poloha se pak porovná a případně se zavolá funkce *setActive()* s parametrem `true` v proměnné `Body` **`nepratele`**`Body` ze třídy Abstraktni_nepratele.

Aktualizace položek probíha obdobně podmínkou for, kde je do třídy Abstraktni_polozky zavolána funkce *update()* s parametrem **`cas`**. Před aktualizací položek, musí být ještě zavolány funkce pro vytvoření položek: *vytvor_voda_2()* a *vytvor_bonus_2()*.

```
//Funkce pro updatování
public void update(float cas)
{
    //Rychlost kalkulace snímků
    world.step(1 / 60f, 6,2);

    //Update vstupů
    uzivatelsky_vstup();

    // Update vstupů - joystick
    uzivatelsky_vstup_joystick();

    //Update Bruna
    bruno.update(cas);

    //Update HUD
    hud.update(cas);

    //Kamera se pousová s polohou hráče
    orthographicCamera.position.x = bruno.brunoBody.getPosition().x;
    //Update kamery
    orthographicCamera.update();
    //Rendrování jen toho, co vidí kamera
    orthogonalTiledMapRenderer.setView(orthographicCamera);

    //Update nepřátel
    for(Abstraktni_nepratele abstraktniNepratele :
vytvoreniSveta.zobrazeniNepratel())
    {
        abstraktniNepratele.update(cas);

        //Probudezení nepřátel, až když je Bruno blízko
        if (abstraktniNepratele.getX() < bruno.getX() + 200 / Hra.korekce)
        {
            abstraktniNepratele.nepratelebody.setActive(true);
        }
    }

    //Vytvoření položek
    vytvor_voda_2();
    vytvor_bonus_2();

    //Update položky - voda
    for(Abstraktni_polozky abstraktniVoda : vodaArray)
        abstraktniVoda.update(cas);
    //Update položky - bonus
    for(Abstraktni_polozky abstraktniBonus : bonusArray)
        abstraktniBonus.update(cas);
}
```

Funkce *dispose()* je vytvořená pro zavírání jednotlivých prvků z Box2D enginu, tím se docílí ušetření operační paměti.

```
//Funkce pro uzavírání
@Override
public void dispose()
{
    world.dispose();
    hud.dispose();
    tiledMap.dispose();
    orthogonalTiledMapRenderer.dispose();
    box2DDebugRenderer.dispose();
}
```

4.3 Obrazovka_konec_hry a Obrazovka_konec_levelu

Třídy Obrazovka_konec_hry a Obrazovka_konec_levelu promítnou po ukončení hry resp. levelu, černou obrazovku s nápisem: Konec hry resp. Konec levelu (Obr.14), dosažené score a nejvyšší score. Kód je u obou tříd stejný, pouze s rozdílem zobrazovaného textu, proto je v kapitole vysvětlena pouze třída Obrazovka_konec_hry.

Obr. 14: Ukázka obrazovky pro konec hry

Obdobně jako předchozí třída, je do této třídy naimplementován prvek `Screen` z pluginy badlogic.gdx a jeho funkce.

```
public class Obrazovka implements Screen
```

Pro promítání je zde vytvořena proměnná `Viewport` **`viewport`**, do které je zavolána funkce *FitViewport(),* s parametry rozměrů hrací plochy a funkcí OrthograpgicCamera(). Následně je vytvořen prvek `stage`, zavoláním do proměnné `Stage` **`stage`**, funkci *Stage()* s parametry právě proměnné **`viewport`** a proměnné **`spriteBatch`**, zavolané ze třídy Hra.
Zobrazení textu je vytvořeno pomocí prvků `Label` **`labelKonecHry`**, **`labelHrejZnovu`**, atd., kde se do úvozovek deklaruje požadovaný zobrazovaný text. Font zobrazovaného písma je vytvořen v externím programu Hiero a implementován do složky assets. V kódu je alokován do proměnné labelStyle, kde je nastavena i barva písma. .Následně je vytvořen prvek `Table` **`table`**, do kterého jsou funkcí *add()* přidány prvky `Label`. Funkce *expandx()* zarovná text v x-vé ose, funkce *row()* oddělí řádek mezi texty a funkce *padTop()* vytvoří mezeru mezi řádky. Po tom, co prvky Label byly vloženy do `Tabel`, musí být ještě `Table` pomocí funkce *addActor()* vložen do `Stage`.
Výpočet score probíhá ve třídě Hud, odkuď je také ukládán do databáze. Pro alokování databáze v téhle třídě je do proměnné `Preferences` **`preferences`**, pomocí funkce *getPreferences()*, alokována databáze a do proměnných `Integer` **`scoreZobrazeni`** a **`nejvyssiScoreZobrazeni`**, je funkcí *getInteger()*, načtena hodnota, která se následně bude zobrazovat.

```
//Konstruktor třídy
public Obrazovka_konec_hry(Game game)
{
    //Napojení na herní svět
    this.game = game;

    //Nastavení kamery na rozměrové veličiny
    viewport = new FitViewport(Hra.sirka, Hra.vyska, new
OrthographicCamera());
    //Inicializace prvku stage
    stage = new Stage(viewport, ((Hra) game).spriteBatch);
    //Nastavení prvku labelstyle, barva písma na bílou
    labelStyle = new Label.LabelStyle(new
BitmapFont(Gdx.files.internal("font2.fnt")), Color.WHITE);
    //Inicializace prvku table
    table = new Table();

    //Hodnoty se, pomocí prvku preferences, ukládádají do databáze, aby mohly
být použity v ostatních třídách
    preferences = Gdx.app.getPreferences("score_databaze");
    scoreZobrazeni = preferences.getInteger("score");
    nejvyssiScoreZobrazeni = preferences.getInteger("nejvyssiScore");

    //Nastavení prvků label
    labelKonecHry = new Label("KONEC HRY", labelStyle);
    labelHrejZnovu = new Label("Stiskem zacnes znovu!", labelStyle);
    labelScore2 = new Label("TVOJE SCORE: ", labelStyle);
    labelScore = new Label(String.format("%04d", scoreZobrazeni), labelStyle);
    labelNejvyssiScore2 = new Label("NEJVYSSI SCORE: ", labelStyle);
    labelNejvyssiScore = new Label(String.format("%04d",
nejvyssiScoreZobrazeni), labelStyle);

    //Nastavení prvku table, row() - odseknutí řádku, padTop() - odseknutí od
horního okraje
    table.add(labelKonecHry).expandX();
    table.row();
    table.add(labelScore2).expandX().padTop(10f);
    table.row();
    table.add(labelScore).expandX();
    table.row();
    table.add(labelNejvyssiScore2).expandX();
    table.row();
    table.add(labelNejvyssiScore).expandX();
    table.row();
    table.add(labelHrejZnovu).expandX().padTop(10f);
    table.center();
    table.setFillParent(true);

    //Nastavení prvku stage
    stage.addActor(table);
}
```

Funkce vytvořená pro rendrování obrazovky: *render()*, pracuje s časovou proměnnou `float` **`cas`**. Podmínkou je pomocí funkce *justTouched()*, zavolané do proměnné `Gdx.`**`input`**, zjištěno jestli hráč stiskl jakoukoliv klávesu. V takovém případě je pomocí funkce *setScreen()* s parametrem new *Obrazovka()*, zavolané do proměnné `Game` **`game`**, spuštěna nová hra.
Do proměnné `Gdx.`**`gl`** jsou zavolány funkce *glClearColor()* a *glClear()*, které obrazovnku vyčistí a nastaví ji černé pozadí.

V posledním kroku je do proměnné `Stage` **`stage`**, zavolána rendrovací funkce *draw()*.

```
//Funkce pro rendrování
@Override
```

```
public void render(float cas)
{
    //zapnutí nové hry po kliku
    if(Gdx.input.justTouched())
    {
        game.setScreen(new Obrazovka((Hra) game));
        dispose();
    }

    Gdx.gl.glClearColor(0, 0, 0, 1);
    Gdx.gl.glClear(GL20.GL_COLOR_BUFFER_BIT);

    stage.draw();
}
```

Je zde i funkce *dispose()*, která pomocí funkce *dispose()* uzavře prvek `Stage` **`stage`**, z důvodu úspory operační paměti.

```
//Funkce pro uzavírání
@Override
public void dispose()
{
    stage.dispose();
}
```

4.4 Hud

Hud je lišta, která se zobrazuje v horní části hrací plochy a slouží k zobrazování aktuálních informací, např. score nebo zbývajícího času na dokončení levelu (Obr.15).

Obr. 15: HUD

HUD má vlastní `OrthographicCameru` i `Viewport`, protože musí být promítán odděleně od hrací plochy.

Prvek `Stage` si můžeme představit jako prázdný box, do kterého vkládáme ostatní prvy, v tomhle případě prvek `Table`. Poté jsou do prvku `Table`, pomocí funkce *add()*, vloženy prvky `Label`, které slouží k zobrazování informací a nastavení zobrazování informací, které je realizováno obdobně jako ve třídě Obrazovka_konec_hry. Výjma funkce *format()*, která nastavuje počet zobrazovaných míst. V posledním kroku jsou funkcí *addActror()* vložen prvek `table` do prvku `stage`.

```
//Konstruktor třídy
public Hud(SpriteBatch spriteBatch)
{
    //Nastavení kamery na rozměrové veličiny
    viewport = new FitViewport(Hra.sirka, Hra.vyska, new
OrthographicCamera());
    //Inicializace prvku stage
    stage = new Stage(viewport, spriteBatch);
    //Nastavení prvku labelstyle, barva písma na bílou
    labelStyle = new Label.LabelStyle(new
BitmapFont(Gdx.files.internal("font.fnt")), Color.WHITE);
    //Inicializace prvku table
    table = new Table();

    cas = 200;
    odpocitavani = 0;
    score = 0;

    //Nastavení prvků label
    casLabel = new Label(String.format("%03d", cas), labelStyle);
    scoreLabel = new Label(String.format("%04d", score), labelStyle);
    levelLabel = new Label("BRUNO", labelStyle);

    //Nastavení prvku table
    table.add(scoreLabel).expandX();
    table.add(levelLabel).expandX();
    table.add(casLabel).expandX();
    table.top();
    table.setFillParent(true);

    //Nastavení prvku stage
    stage.addActor(table);
}
```

Funkce *pridej_score()* slouží k přičítání bodů, která hráč během hry získává. Pracuje s proměnnou `int` **`pocet`**, která je přičítána k proměnné `int` **`score`**. Následně je zde pomocí funkce *setText()* nastaven prvek `Label` **`scoreLabel`**, tak aby zobrazoval text na 4 číslice.

Do proměnné `Preferences` **`preferences`** je funkcí *getPreferences()* vytvořena databáze, do je funkcí *putIteger()* vložena hodnota aktuálního score. Z databáze je funkcí *getInteger()* vytáhnuta hodnota nejvyššího score a porovnána s aktuální hodnotou. V případě, že aktuální hodnota je vyšší než předchozí, je uložena jako nejvyšší.

```
//Funkce pro přidání score
public static void pridej_score(int pocet)
{
    //K proměnné se přičtě hodnota score, která se zavolá do funkce
    score += pocet;
    //Zobrazování score, počst míst
    scoreLabel.setText(String.format("%04d", score));

    //Hodnota score se, pomocí prvku preferences, ukládá do databáze, aby
mohla být použita v ostatních třídách
    preferences = Gdx.app.getPreferences("score_databaze");
    preferences.putInteger("score", score);

    //Výpočet a uložení nejvyššího score do databáze
    nejvyssiScore = preferences.getInteger("nejvyssiScore", 0);
    if(score > nejvyssiScore)
    {
        preferences.putInteger("nejvyssiScore", score);
    }
    preferences.flush();
}
```

HUD obsahuje i odpočet času, za který musí být level dokončen. Pro tento účel slouží funkce *update()*, pracující s časovou proměnnou `float` **`cas2`**. Kód pro odpočet je vytvořen pomocí proměnné `float` **`odpocitavani`**(deklarované na hodnotu 0), ke které je přičítána proměnná `float` **`cas`**. Následně pomocí podmínky s parametrem **`odpocitavani`** >= 1, je každou sekundu proměnná **`cas`**(deklarovaná na hodnotu 200) snížena o 1. Prvek pro zobrazování hodnot `Label` **`casLabel`** je pomocí funkce *setText()* naformátován, aby zobrazoval čas na 3 číslice. V posledním kroku je proměnná **`odpocitavni`** vynulována.

```
//Funkce pro updatování
public void update(float cas2)
{
    odpocitavani += cas2;

    //Cyklus pro odečítání času na splnění levelu
    if(odpocitavani >= 1)
    {
        this.cas--;
        //Zobrazování, počet zobrazovaných míst
        casLabel.setText(String.format("%03d", this.cas));
        odpocitavani = 0;
    }
}
```

V poslední kroku je opět prvek `Stage` uzavřen ve funkci *dispose()*.

```
//Funkce pro uzavírání
@Override
public void dispose()
{
    stage.dispose();
}
```

4.5 Bruno

Třída Bruno je oděděná od prvku `Sprite`, který je součástí pluginu libGDX.

```
public class Bruno extends Sprite
```

V konstruktoru je prvně alokován herní svět pomocí proměnné `World` **`world`** a funkce *getWorld()*, zavolané do proměnné `Obrazovka` **`brunoObrazovka`**. Bruno se může nacházet v několika stavech. Kvůli tomu jsou vytvořeny pomocné proměnné `brunoStav` **`brunoSoucasnyStav`** a **`brunoPredchoziStav`**, kterým je přiřazen stav stání. Je zde také vynulována proměnná `float` **`stavovyCas`** a zavolány funkce *definiceBruna()* a *animace()*.

```
//Konstruktor třídy
public Bruno(Obrazovka brunoObrazovka)
{
    //Napojení na herní svět
    this.world = brunoObrazovka.getWorld();
    this.obrazovka = brunoObrazovka;

    brunoSoucasnyStav = brunoPredchoziStav = brunoStav.stani;

    stavovyCas = 0;

    definiceBruna();

    animace();
}
```

Animace postavy je vytvořena ve funkce *animace()*. Prvně je vytvořen prvek `com.badlogic.gdx.utils.Array<TextureRegion>` **`brunoAnimace`**, do kterého je přidán soubor s textury, vytvořený v programu GDX Texture Packer. Jednotlivé animace: stání, běhu, skoku, růstu a umírání mají obdobný kód. Pomocí podmínky for se vytvoří logika pro jednolivé snímky animace a poté jsou funkcí *add()* přidány do **`brunoAnimace`**. Pomocí funkce *TextureRegion()* jsou alokovány patřičné animace a pixelová rozloha i poloha jednolivých snímků. Proměnné typu `Animation<TextureRegion>` slouží pro nastavení rychlosti animace. Po každem animačním cyklu je potřeba pomocí funkce *clear()* zavoláne do proměnné **`brunoAnimace`** vyčistit proměnou. Animace růstu není vytvořena pomocí podmínky for, jde totiž jen o dvojité vystřídání textury malého Bruna a velkého.

```
//Funkce pro animace
public void animace()
{
    brunoAnimace = new com.badlogic.gdx.utils.Array<TextureRegion>();

    for (int i = 1; i < 4; i++)
    {
        brunoAnimace.add(new
TextureRegion(obrazovka.getTextureAtlas().findRegion("BrunoMaly"), i * 17, 0,
17, 16));
        brunoAnimaceBeh = new Animation<TextureRegion>(0.1f, brunoAnimace);
    }
    brunoAnimace.clear();

    for (int i = 1; i < 4; i++)
    {
        brunoAnimace.add(new
TextureRegion(obrazovka.getTextureAtlas().findRegion("BrunoVelky"), i * 17, 0,
17, 32));
        brunoAnimaceBehVelky = new Animation<TextureRegion>(0.1f,
brunoAnimace);
    }
    brunoAnimace.clear();

    brunoAnimace.add(new
TextureRegion(obrazovka.getTextureAtlas().findRegion("BrunoVelky"), 85, 0, 17,
32));
    brunoAnimace.add(new
TextureRegion(obrazovka.getTextureAtlas().findRegion("BrunoVelky"), 0, 0, 17,
32));
    brunoAnimace.add(new
TextureRegion(obrazovka.getTextureAtlas().findRegion("BrunoVelky"), 85, 0, 17,
32));
    brunoAnimace.add(new
TextureRegion(obrazovka.getTextureAtlas().findRegion("BrunoVelky"), 0, 0, 17,
32));
    brunoAnimaceRust = new Animation<TextureRegion>(0.2f, brunoAnimace);
    brunoAnimace.clear();

    brunoAnimaceSkok = new
TextureRegion(obrazovka.getTextureAtlas().findRegion("BrunoMaly"), 85, 0, 17,
16);

    brunoAnimaceSkokVelky = new
TextureRegion(obrazovka.getTextureAtlas().findRegion("BrunoVelky"), 119, 0,
17, 32);

    brunoAnimaceStani = new
TextureRegion(obrazovka.getTextureAtlas().findRegion("BrunoMaly"), 0, 0, 17,
16);

    brunoAnimaceStaniVelky = new
TextureRegion(obrazovka.getTextureAtlas().findRegion("BrunoVelky"), 0, 0, 17,
32);

    brunoAnimaceSmrt = new
TextureRegion(obrazovka.getTextureAtlas().findRegion("BrunoMaly"), 170, 0, 17,
16);

    //Definování reálných okrajů textur
    setBounds(0, 0, 16 / Hra.korekce, 16 / Hra.korekce);

    setRegion(brunoAnimaceStani);
}
```

Vzhledem k tomu, že Bruno bude během hry měnit svoje stavy, musí být vytvořena funkce `TextureRegion` *animace2()*. Funkce pracuje s časovou proměnnou `float` **`cas`**. Do proměnné **`brunoSoucasnyStav`** je zavolána funkce *aktualniStav(),* která určí jeho současný stav a pomocí switch se na základě hodnoty proměnné určí hodnota proměnné `TextureRegion` **`textureRegion`**.

Animace běhu, růstu, skákání a padání musí obsahovat navíc kód pro situaci, kdy je Bruno velký. Pomocí znaku ? se zkontroluje jestli má proměnná `boolean` **`brunoJeVelky`** hodnotu `true`, jestli ano, spustí se animace velkého Bruna, pokud ne malého.

Běh Bruna se odehrává buď v plusových hodnotách x nebo záporných. Je zde tedy podmínka, ve kterém se funkcí *getLinearVelocity()*, zavolanou do proměnné `Body` **`brunoBody`**, zjistí X-ové hodnoty Bruna. Funkcí *isFlipX(),* zavolanou do proměnné **`textureRegion`**, se zjistí na kterou stranu je textura otočena. Funkce *flip(true, false),* kde první parametr určuje rendrování v X-ové ose a druhý v Y-ové, se definuje rendrování textury ve směru X.

```
//Funkce pro animace(jednotlivé stavy)
public TextureRegion animace2(float cas)
{
    TextureRegion textureRegion;

    brunoSoucasnyStav = aktualniStav();

    switch (brunoSoucasnyStav)
    {
        case beh:
            //Když je Bruno velký animace pro velký běh, jinak pro malý
            textureRegion = brunoJeVelky ?
brunoAnimaceBehVelky.getKeyFrame(stavovyCas, true) :
brunoAnimaceBeh.getKeyFrame(stavovyCas, true);
            break;

        case skok:
            //Když je Bruno velký animace pro velký skok, jinak pro malý
            textureRegion = brunoJeVelky ? brunoAnimaceSkokVelky :
brunoAnimaceSkok;
            break;

        case stani:
        default:
            //Když je Bruno velký animace pro velké stání, jinak pro malé
            textureRegion = brunoJeVelky ? brunoAnimaceStaniVelky :
brunoAnimaceStani;
            break;

        case rust:
            textureRegion = brunoAnimaceRust.getKeyFrame(stavovyCas);

            //Vypnutí animace růstu, ikdyž není looping
            if(brunoAnimaceRust.isAnimationFinished(stavovyCas))
                brunoRust = false;
            break;

        case smrt:
            textureRegion = brunoAnimaceSmrt;
            break;
    }

    //Zrcadlové promítnutí animace podle směru pohybu
    if(brunoBody.getLinearVelocity().x < 0 && textureRegion.isFlipX() ==
false)
    {
        textureRegion.flip(true, false);
    }
    else if(brunoBody.getLinearVelocity().x > 0 && textureRegion.isFlipX()  ==
true)
    {
        textureRegion.flip(true, false);
    }

    //Pokud je současný shodný s předchozím --> (stavovyCas + cas) = 0
    stavovyCas = brunoSoucasnyStav == brunoPredchoziStav ? stavovyCas + cas :
0;

    brunoPredchoziStav = brunoSoucasnyStav;

    return textureRegion;
}
```

Funkce *definice_bruna()* slouží k definování vlastností Bruna. Prvně je do proměnné `BodyDef` **`brunoDefiniceBody`** zavolána funkce *BodyDef()* a následně do **`brunoDefiniceBody`** nastavena poloha Bruna a nadefinování jeho základních vlastností jako DynamicBody. Poté se do proměnné `Body` **`brunoBody`** funkcí *createBody()*, s parametrem **`brunoDefiniceBody`**, zavolané do proměnné **`world`**, přiřadily definované vlastnosti Bruna.

K prvku `Body` jsou následně nastaveny jeho vlastnosti pomocí `Fixtur`. Prvně jsou zde vytvořeny proměnné `FixtureDef` **`brunoDefiniceFixture`** pro zavolání funkce *FixtureDef()* a `CircleShape` **`brunoTelo`** pro zavolání funkce *CircleShape()*. Pro proměnnou **`brunoTelo`**, se funkcí *setRadius()*, nastavil rádius na hodnotu 6, tím byl vytvořen kruhový objekt, který interaguje s okolím. K **`brunoDefiniceFixture`** jsou ze třídy Hra nastaveny `categoryBits`, neboli bity, které Bruna zastupují a `maskBits`, neboli Bity s kterými Bruno interaguje. Na závěr se pomocí funkce *createFixture()*, s paramtrem **`brunoDefiniceFixture`**, zavolané do **`brunoBody`**, vytvořila fixtura do **`brunoBody`**.

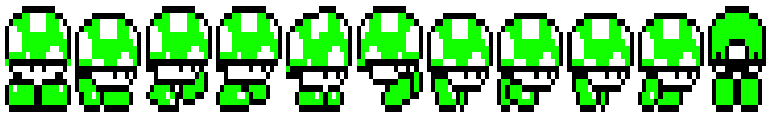
Jelikož Bruno interaguje svojí hlavou jinak než zbytkem těla, je potřeba jeho hlavě vytvořit vlastní prvek pomocí proměnné `PolygonShape` **`brunoHlava`**, který jde chápat jako trojúhelník mezi třemi body, které jsou nastaveny pomocí prvku *Vector2()*. Poté je prvek, funkcí *createFixture()* s parametrem **`brunoDefiniceFixture`**, přidán do **`brunoBody`**.

```
//Funkce pro definování vlastností
public void definice_bruna()
{
    brunoDefiniceBody = new BodyDef();
    brunoDefiniceFixture = new FixtureDef();
    brunoTelo = new CircleShape();
    brunoHlava = new PolygonShape();

    //Definice typu Body
    brunoDefiniceBody.type = BodyDef.BodyType.DynamicBody;
    //Definice polohy
    brunoDefiniceBody.position.set(32 / Hra.korekce, 32 / Hra.korekce);
    //Vytvoření Body
    brunoBody = world.createBody(brunoDefiniceBody);

    //Vytvoření kruhového Body
    brunoTelo.setRadius(6 / Hra.korekce);
    brunoDefiniceFixture.shape = brunoTelo;

    //Definice bitů, které objekt zastupují
    brunoDefiniceFixture.filter.categoryBits = Hra.bruno_bit;
    //Definice bitů, se kterými je objekt v kontaktu
    brunoDefiniceFixture.filter.maskBits = Hra.zeme_bit | Hra.mince_bit |
Hra.cihla_bit | Hra.nepratele_bit | Hra.objekty_bit | Hra.hlava_nepritele_bit
| Hra.vec_bit| Hra.vor_bit | Hra.okraje_bit | Hra.vor_hlava_bit |
Hra.bonus_bit;

    //Vytvoření Fixture
    brunoBody.createFixture(brunoDefiniceFixture).setUserData(this);

    //Vytvoření trojúhleníku, který alkokuje hlavu
    brunoHlava2 = new Vector2[3];
    brunoHlava2[0] = new Vector2(-3,5).scl(1 / Hra.korekce);
    brunoHlava2[1] = new Vector2(3,5).scl(1 / Hra.korekce);
    brunoHlava2[2] = new Vector2(0,3).scl(1 / Hra.korekce);
    brunoHlava.set(brunoHlava2);
    brunoDefiniceFixture.shape = brunoHlava;

    //Definice bitů pro hlavu
    brunoDefiniceFixture.filter.categoryBits = Hra.hlava_bruna_bit;

    //brunoDefiniceFixture.isSensor = true;

    //Vytvoření Fixture
    brunoBody.createFixture(brunoDefiniceFixture).setUserData(this);
}
```

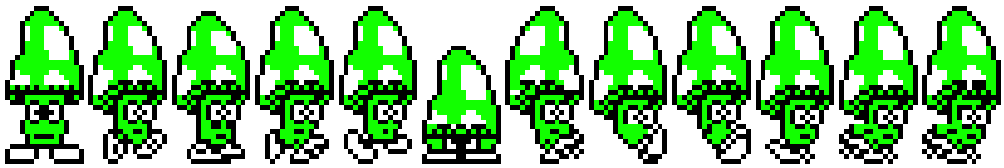
Funkce *definice_velkeho_bruna()* je obdobná funkci *definice_bruna()*. Nejprve se do proměnné `Vector2` **`brunoPozice`**, funkcí *getPosition()*, zavolanou do proměnné `Body` **`brunoBody`**, zjistí poloha Bruna. Následně se funkcí *destroyBody()* s parametrem **`brunoBody`**, zavolanou do proměnné `World` **`world`**, se malý Bruno vymaže z herního světa.

Vzhledem k tomu, že když se zvětší Brunova grafická textura, musí se změnit aji poloha jeho `Body`, jinak by velký Bruno byl propadlí do země. Proto je do proměnné `BodyDef` **`brunoDefiniceVelkeho`** s parametrem **`position`** zavolána funkce *set()*, ve které se jeho současná poloha: **`brunoPozice`** funkcí *add()* vycentruje.

Musí být upraverna aji proměnná `CircleShape` **`brunoTelo`**. Do proměnné se zavolá funkce *setPosition()*, s vnořenou funkcí *Vector2()*, která vytvoří opět kruhový objekt. Objekt je vycentrován pod stávající `CircleShape`, tak aby pokryl rozlohy celého Bruna. Grafická textury malého Bruna jsou zobrazeny na obr.16 a velkého Bruna na obr. 17.



Obr. 16: Malý Bruno



Obr. 17: Velký Bruno

```
//Funkce pro definování vlastností velkého Bruna
public void definice_velkeho_bruna()
{
    //Do proměnné se nahraje současný vektor pozice Bruna
    brunoPozice = brunoBody.getPosition();
    //Body Bruna je zničeno
    world.destroyBody(brunoBody);

    brunoDefiniceVelkeho = false;

    brunoDefiniceBody = new BodyDef();
    brunoDefiniceFixture = new FixtureDef();
    brunoTelo = new CircleShape();

    //Definice typu Body
    brunoDefiniceBody.type = BodyDef.BodyType.DynamicBody;
    //korekce toho, aby Bruno nebyl po vyrosteni propadlej do zeme
    brunoDefiniceBody.position.set(brunoPozice.add(0, 10 / Hra.korekce));
    //Vytvoření Body
    brunoBody = world.createBody(brunoDefiniceBody);

    //Vytvoření kruhového Body
    brunoTelo.setRadius(6 / Hra.korekce);
    brunoDefiniceFixture.shape = brunoTelo;

    //Definice bitů, které objekt zastupují
    brunoDefiniceFixture.filter.categoryBits = Hra.bruno_bit;
    //Definice bitů, se kterými je objekt v kontaktu
    brunoDefiniceFixture.filter.maskBits = Hra.zeme_bit | Hra.mince_bit |
Hra.cihla_bit | Hra.nepratele_bit | Hra.objekty_bit | Hra.hlava_nepritele_bit
| Hra.vec_bit | Hra.okraje_bit | Hra.vor_hlava_bit| Hra.bonus_bit;

    //Vytvoření Fixture
    brunoBody.createFixture(brunoDefiniceFixture).setUserData(this);

    //Vycetnrování, aby nebyl prodadlý do země
    brunoTelo.setPosition(new Vector2(0, -14 / Hra.korekce));

    //Vytvoření Fixture
    brunoBody.createFixture(brunoDefiniceFixture).setUserData(this);

    //Vytvoření trojúhleníku, který alkokuje hlavu
    brunoHlava2 = new Vector2[3];
    brunoHlava2[0] = new Vector2(-3,5).scl(1 / Hra.korekce);
    brunoHlava2[1] = new Vector2(3,5).scl(1 / Hra.korekce);
    brunoHlava2[2] = new Vector2(0,3).scl(1 / Hra.korekce);
    brunoHlava.set(brunoHlava2);
    brunoDefiniceFixture.shape = brunoHlava;

    //Definice bitů pro hlavu
    brunoDefiniceFixture.filter.categoryBits = Hra.hlava_bruna_bit;

    //Vytvoření Fixture
    brunoBody.createFixture(brunoDefiniceFixture).setUserData(this);
}
```

Funkce *definice_opet_maleho_bruna()* slouží k nastavení prvku `Body`, po tom co je zničen velký Bruno a vznikne opět Bruno v normální velikosti. Tato třída je logicky opět obdobná třídě *definice_bruna(),* s tím rozdílem, že musí zničit velkého Bruna a na jeho místě vytvořit nevyrostlého. Pro tyto účely je zde tedy vytvořena proměnná `Vector2` **`brunoPozice`**, do které je funkcí *getPosition(),* zavolanou do proměnné `Body` **`brunoBody`**, načtena poloha `Body`. Následně do proměnné `Bodydef` **`brunoDefiniceBody`**, s parametrem **`position`**, je funkcí *set(),* s proměnnou **`brunoPozice`**, alokována poloha zničeného Bruna.

Velký Bruno je zničen zničením jeho Body. Do proměnné `World` **`world`** je zavolána funkce *destroyBody()* s parametrem `Body` **`brunoBody`**.

Musí zde být nastavena i proměnná `boolean` **`brunoOpetMaly`** na hodnotu false.

```
//Funkce pro definování vlastností opět malého Bruna
public void definice_opet_maleho_bruna()
{
    //Do proměnné se nahraje současný vektor pozice Bruna
    brunoPozice = brunoBody.getPosition();
    //Body Bruna je zničeno
    world.destroyBody(brunoBody);

    brunoOpetMaly = false;

    brunoDefiniceBody = new BodyDef();
    brunoTelo = new CircleShape();
    brunoDefiniceFixture = new FixtureDef();

    //Definice typu Body
    brunoDefiniceBody.type = BodyDef.BodyType.DynamicBody;
    //Definice polohy podle zničeného Body
    brunoDefiniceBody.position.set(brunoPozice);
    //Vytvoření Body
    brunoBody = world.createBody(brunoDefiniceBody);

    //Vytvoření kruhového Body
    brunoTelo.setRadius(6 / Hra.korekce);
    brunoDefiniceFixture.shape = brunoTelo;

    //Definice bitů, které objekt zastupují
    brunoDefiniceFixture.filter.categoryBits = Hra.bruno_bit;
    //Definice bitů, se kterými je objekt v kontaktu
    brunoDefiniceFixture.filter.maskBits = Hra.zeme_bit | Hra.mince_bit |
Hra.cihla_bit | Hra.nepratele_bit | Hra.objekty_bit | Hra.hlava_nepritele_bit
| Hra.vec_bit | Hra.vor_bit | Hra.okraje_bit | Hra.vor_hlava_bit|
Hra.bonus_bit;

    //Vytvoření Fixture
    brunoBody.createFixture(brunoDefiniceFixture).setUserData(this);

    //Vytvoření trojúhleníku, který alkokuje hlavu
    brunoHlava2 = new Vector2[3];
    brunoHlava2[0] = new Vector2(-3,5).scl(1 / Hra.korekce);
    brunoHlava2[1] = new Vector2(3,5).scl(1 / Hra.korekce);
    brunoHlava2[2] = new Vector2(0,3).scl(1 / Hra.korekce);
    brunoHlava.set(brunoHlava2);
    brunoDefiniceFixture.shape = brunoHlava;

    //Definice bitů pro hlavu
    brunoDefiniceFixture.filter.categoryBits = Hra.hlava_bruna_bit;

    //brunoDefiniceFixture.isSensor = true;

    //Vytvoření Fixture
    brunoBody.createFixture(brunoDefiniceFixture).setUserData(this);
}
```

Funkce `brunoStav` *aktualni_stav()* slouží pro získání aktuálního stavu Bruna. Např. stav běhu je zjištěn zavoláním funkce *getLinearVelocity()* do proměnné `Body` **brunoBody**. Jestli rychlost Bruna v X-ové ose není rovna 0, Bruno se musí zákonitě pohybovat. Obdobně jsou zjištěny i stavy skok a pad. Ostatní stavy nastanou v případě, že jejich příslušná proměná typu `boolean` nabude hodnoty `true`.

```
//Funkce pro získání současného stavu Bruna
public brunoStav aktualni_stav()
{
    //Aby bruno běžěl, nesmí být rychlost v X rovna 0
    if (brunoBody.getLinearVelocity().x != 0 &&
brunoBody.getLinearVelocity().y <= 0 && brunoBody.getLinearVelocity().y >= 0
&& !brunoJeMrtvy)
        return brunoStav.beh;

    //Když bruno skáče, je jeho rychlost v Y nenulová
    else if (brunoBody.getLinearVelocity().y > 0 ||
(brunoBody.getLinearVelocity().y < 0) && !brunoJeMrtvy)
        return brunoStav.skok;

    //Stav se mění podle proměnné boolean
    else if(brunoRust)
        return brunoStav.rust;

    //Stav se mění podle proměnné boolean
    else if(brunoJeMrtvy)
        return brunoStav.smrt;

    else
        return brunoStav.stani;
}
```

Funkce *uder(),* pracuje s proměnnou `Abstraktni_nepratele` **abstraktniNepratele**. Funkce je volána při kontaktu Bruna s nepřítelem. Při tomto kontaktu může nastat hned několik situací, které jsou zde zohledněny.

V podmínce se zjistí, jestli nepřítel, kterého Bruno udeřil je Klungo, tzn. instance **abstraktniNepratele** je Klungo. Pomocí funkce *ziskejSoucasnyStav()* je zjištěn aktuální stav Klunga a následně je porovnán s jeho stavem stojícího krunýře. V takovémto případě je zavolána funkce *kopnuti()* ze třídy Klungo, funkcí *getX()* je zjištěna X-ové souřadnice Bruna a Klunga. Následně jsou porovnány, čímž se určí z které strany Bruno Klunga kopnul a podle toho je Klungovi udělana kladná nebo záporná rychlost v ose x.

V else větvi je vnořená podmínka, která zohledňuje situaci, kdy Bruno změní svůj stav z velkého na malého. Zjistí se jestli má proměnná `boolean` **brunoJeVelky** hodnotu `true`, tzn. Bruno je velký, pokud ano je proměnná nastavena na hodnotu false a proměnné `boolean` **brunoOpetMaly** je nastavena hodnota `true`. Je potřeba zavolat funci *setBounds()*, ve které je zavolána funkce *getHeight()*, která je podělena dvěma, aby se reálné rozměry vrátily na velikost nevyrostlého Bruna. Následně je zpuštěn zvukový soubor obdobně jako kdykoliv před tím.

Ve větvi else se řeší situace, kdy Bruno zemře. Pomocí funkce *stop()* je vypnut audio soubor, který hraje v pozadí hry a současně. Funkcí *play()*, je zpuštěn zvuk, který doprovází Brunovo zabití. Proměnná `boolean` **brunoJeMrtvy** je nastavena na hodnotu `true`. Po tom, co Bruno zemře, nesmí interagovat s okolím, proto jsou jeho `maskBits`, opět pomocí prvku filter, nastaveny na speciální bity: `prazdny_bit`. Do proměnné `Body` **brunoBody** je zavolána funkce *applyLinearImpulse()* s vnořenou funkcí *Vector2()*, která aplikuje na tělo Bruna skok v Y-ové ose.

```
//Funkce pro kolizi s nepřítelem
public void uder(Abstraktni_nepratele abstraktniNepratele)
{
    //Pokud je nepřítel Klungo: stojící krunýř, je podle strany, ze které ho
Bruno kontaktoval, odkopnut
    if(abstraktniNepratele instanceof Klungo && ((Klungo)
abstraktniNepratele).ziskejSoucasnyStav() == klungoStav.stojici_krunyr)
    {
        ((Klungo) abstraktniNepratele).kopnuti(this.getX() <=
abstraktniNepratele.getX() ? Klungo.napravo : Klungo.nalevo);
    }
    else
    {
        //Když je bruno velký a koliduje s nepřítelem, mění se na malého
        if(brunoJeVelky)
        {
            //Spuštění zvukového souboru
            Hra.assetManager.get("audio/zvuky/zmenseni.wav",
Sound.class).play();

            //Definování reálných okrajů textur, výška podělena dvěma aby se
Bruno vrátil na původní velikost
            setBounds(getX(), getY(), getWidth(), getHeight() / 2);

            brunoJeVelky = false;
            brunoOpetMaly = true;
        }
        else
        {
            brunoJeMrtvy = true;

            //Body je při smrti udělen vektor výskoku
            brunoBody.applyLinearImpulse(new Vector2(0, 3f),
brunoBody.getWorldCenter(), true);

            //Vypnutí zvukového souboru
            Hra.assetManager.get("audio/hudba/hudba1.mp3",
Music.class).stop();
            //Spuštění zvukového souboru
            Hra.assetManager.get("audio/zvuky/smrt.wav", Sound.class).play();

            //Nastavení bitů, se kterými objekt koliduje na prázdnou hodnotu
            brunoFilter = new Filter();
            brunoFilter.maskBits = Hra.prazdny_bit;
            for (Fixture fixture : brunoBody.getFixtureList())
                fixture.setFilterData(brunoFilter);
        }
    }
}
```

Funkce *rust()* slouží, po kontaktu s bonusovou položkou: láhve s vodou, ke zvětšení Bruna. Toto zvětšení prakticky způsobí jeden bonusový život navíc, protože když je Bruno velký a zemře, tak nekončí hra, ale Bruno se zmenší do původní velikosti.

Funkce nastaví tři proměnné typu `boolean`: **`brunoDefinicevelkeho`**, **`brunoRust`** a **`brunoJeVelky`** na hodnotu `true`. Je zde také funkce *setBounds()*, sloužící k definování reálných okrajů, kde je její hodnota pro výšku vynásobena dvěma.

```
//Funkce pro zvětšení Bruna
public void rust()
{
    //Spuštění zvukového souboru
    Hra.assetManager.get("audio/zvuky/zvetseni.wav", Sound.class).play();

    //Funkce pro definování reálných okrajů textur, musí být dvojnásobně
zvětšena
    setBounds(getX(), getY(), getWidth(), getHeight() * 2);

    brunoDefiniceVelkeho = true;

    brunoRust = true;

    brunoJeVelky = true;
}
```

Funkce *update()*, jako všechny funkce tohoto typu, pracuje s časovou proměnou `float` **`cas`**. V podmínce se pomocí proměnné `boolean` **`brunoJeVelky`** zjistí, jestli je Bruno velký a aktualizuje se poloha grafických textur jeho `Body`, pomocí funkce *setPosition()*. Ve funkci *setposition()* dochází i k vycetntrování do středu `Body`, vydělením jeho pozice polovinou velikosti rozměru. V podmínce else je zohledněna obdobná situace pro nevyrostlého Bruna.

Funkce obsahuje ještě dvě podmínky, první na základě hodnoty proměnné `boolean` **`brunoDefiniceVelkeho`** volá funkci *definice_velkeho_bruna()* a druhý na základě hodnoty proměnné `boolean` **`brunoOpetMaly`** volá funkci *drefinice_opet_maleho_bruna()*.

```
//Funkce pro updatování
public void update(float cas)
{
    //Cyklus pro spuštění funkce, pro definici velkého Bruna
    if(brunoDefiniceVelkeho)
        definice_velkeho_bruna();

    //Cyklus pro spuštění funkce, pro redefinici Bruna
    if(brunoOpetMaly)
        definice_opet_maleho_bruna();

    //Nastavení pozice - rozměry podělené dvěma z důvodu vycentrování
    if(brunoJeVelky)
        setPosition(brunoBody.getPosition().x - getWidth() / 2,
brunoBody.getPosition().y - getHeight() / 2 - 6 / Hra.korekce);
    else
        setPosition(brunoBody.getPosition().x - getWidth() / 2,
brunoBody.getPosition().y - getHeight() / 2);

    //Nastavení animace
    setRegion(animace2(cas));
}
```

4.6 Třída Abstratni_nepratele

Třída Abstraktni_nepratele je odděděna od prvku `Sprite`, implementovaného z pluginu libGDX.

```
public abstract class Abstraktni_nepratele extends Sprite
```

Konstruktor třídy je pomocí proměnných `World` **`world`** a `Obrazovka` **`obrazovka`** napojen na herní svět. Funkce má také zadané dva parametry typu `float`: **`x`** a **`y`**, ke kterým je přidána funkce *setPosition()*, pomocí které je definována poloha nepřítele.

Jsou zde také zavolány funkce *definice_nepritele()* a *animace()*.

```
//Konstruktor třídy
public Abstraktni_nepratele(Obrazovka nepratele0brazovka, float x, float y)
{
    //Napojení na herní svět
    this.world = nepratele0brazovka.getWorld();
    this.obrazovka = nepratele0brazovka;

    //Funkce deklarující polohu
    setPosition(x, y);

    definice_nepratele();

    animace();
}
```

Funkce *zpetna_rychlost()* udělí reverzní pohyb jakémukoliv nepříteli. Jde tedy o to aby nepřítel nezůstal po kontaktu buď s nějakým objektem, nebo s dalším nepřítelem stát na jednom místě, ale začal vykonávat obrácený pohyb. Funkce pracuje s dvěma proměnnými typu `boolean` **`x`** a **`y`**. Obě proměnné jsou použity jako podmínky, ve kterých se obrátí znaménko pro proměnnou `Vector2` **`rychlostNepratele`**.

```
//Funkce pro reverzní pohyb
public void zpetna_rychlost(boolean x, boolean y)
{
    if(x)
        rychlostNepratele.x = -rychlostNepratele.x;
    if(y)
        rychlostNepratele.y = -rychlostNepratele.y;
}
```

Dále jsou zde vytvořeny funkce, které bude následně naimplementovány do dceřiných tříd.

```
//Funkce pro definování vlastností
protected abstract void definice_nepratele();

//Funkce pro animaci
protected abstract void animace();

//Funkce pro updatování
public abstract void update(float cas);

//Funkce pro zničení nepřítele, když mu Bruno skočí na hlavu
public abstract void skok_na_hlavu(Bruno bruno);

//Funkce pro kolizi s nepřítelem
public abstract void kolize_nepritele(Abstraktni_nepratele
abstraktniNepratele);
```

4.6.1 Balsac

Balsac (Obr.18) je nepřítel, pohybující se v X-ové ose, který je zničitelný skokem na hlavu.

Obr. 18: Balsac

V konstruktoru dochází na napojení na herní svět obdobně jako v předchozích případech. Je zde aji nadefinován směr pohybu Balsaca v X-ové ose.

```
//Konstruktor třídy
public Balsac(Obrazovka obrazovkaBalsac, float x, float y)
{
    //Napojení na herní svět
    super(obrazovkaBalsac, x, y);
    //Deklarace rychlosti a směru pohybu
    rychlostNepratele = new Vector2(1, 0);
}
```

Funkce pro animaci Balsaca: animace je obdobná jako ve třídě Bruno.

```
//Funkce pro animace
protected void animace()
{
    balsacAnimace2 = new Array<TextureRegion>();

    //Cyklus pro promítnutí snímků animace
    for(int i = 0; i < 2; i++)
    {
        balsacAnimace2.add(new
TextureRegion(obrazovka.getTextureAtlas().findRegion("Balsac"), i * 16, 0, 16,
16));
        balsacAnimace = new
com.badlogic.gdx.graphics.g2d.Animation<TextureRegion>(0.4f, balsacAnimace2);
    }

    //Definování reálných okrajů textur
    setBounds(getX(), getY(), 16 / Hra.korekce, 16 / Hra.korekce);
}
```

Třída *definice_nepritele()* je naimplementována ze třídy Abstraktni_nepratele. A je obdobná třídě Bruno.

Je zde zadefinováno `Body` jako `DynamicBody` a pomocí proměnné `CircleShape` **`balsacTelo`** je nastaven tvar a vlastnosti Fixtury. Filtr pro categoryBits je alokován k vlastnímu bitu, `nepratele_bit` a filtr pro `maskBits` jsou nastaven ke všem objektům, se kterými má Balsac reagovat.

Balsac je zničitelný pouze skokem na hlavu, proto jsou zde vytvořeny proměnné `PolygonShape` **`balsacHlava`** a `Vector2[]` **`balsachlava2`**, pomocí které je bodově určen rozměr hlavy. K proměnné `FixtureDef` **`balsacDefiniceFixture`** je definován parametr `restitution`, který způsobí odraz Bruna od hlavy Balsaca. Hlavě je také přiřazen vlastní filtr pro `categoryBits`, `hlava_nepritele_bit`.

```
//Funkce pro definování vlastností
@Override
protected void definice_nepratele()
{
    balsacDefiniceBody = new BodyDef();
    balsacDefiniceFixture = new FixtureDef();
    balsacTelo = new CircleShape();
    balsacHlava = new PolygonShape();

    //Definice typu Body
    balsacDefiniceBody.type = BodyDef.BodyType.DynamicBody;
    //Definice polohy
    balsacDefiniceBody.position.set(getX(), getY());
    //Vytvoření Body
    neprateleBody = world.createBody(balsacDefiniceBody);

    //Vytvoření kruhového Body
    balsacTelo.setRadius(6 / Hra.korekce);
    balsacDefiniceFixture.shape = balsacTelo;

    //Definice bitů, které objekt zastupují
    balsacDefiniceFixture.filter.categoryBits = Hra.nepratele_bit;

    //Definice bitů, se kterými je objekt v kontaktu
    balsacDefiniceFixture.filter.maskBits = Hra.zeme_bit | Hra.mince_bit |
Hra.cihla_bit | Hra.bruno_bit | Hra.nepratele_bit | Hra.objekty_bit |
Hra.okraje_bit;

    //Vytvoření Fixture
    neprateleBody.createFixture(balsacDefiniceFixture).setUserData(this);

    //Vytvoření trojúhleníku, který alkokuje hlavu
    balsacHlava2 = new Vector2[3];
    balsacHlava2[0] = new Vector2(-5,7).scl(1 / Hra.korekce);
    balsacHlava2[1] = new Vector2(5,7).scl(1 / Hra.korekce);
    balsacHlava2[2] = new Vector2(0,3).scl(1 / Hra.korekce);
    balsacHlava.set(balsacHlava2);
    balsacDefiniceFixture.shape = balsacHlava;

    //Definice intenzity odrazu po skuku na hlavu
    balsacDefiniceFixture.restitution = 0.3f;

    //Definice bitů pro hlavu
    balsacDefiniceFixture.filter.categoryBits = Hra.hlava_nepritele_bit;

    //Vytvoření Fixture
    neprateleBody.createFixture(balsacDefiniceFixture).setUserData(this);

    //Objekt se aktvivuje až v blízkosti hráče, z důvodu úspory paměti
    neprateleBody.setActive(false);
}
```

Funkce *kolize_nepritele()* řeší kontakt Balsaca s ostatními nepřáteli. Ověří, jestli je nepřítel Klungo, tzn. instance třídy Abstraktni_nepratele je Klungo a současně je zjištěn jestli je aktuální stav Klunga pohybující se krunýř. V tomto případě Balsac Klungem zničen a proměnná `boolean` **`balsacZnicit`** je nastavena na hodnotu true, je spuštěň zvukový soubor, navýšeno score a předefinovány `maskBits` na `prazdny_bit`. V opačném případě, tzn. ve větvi else je funkce *zpetna_rychlost()*, která udělí nepřátelům reverzní rychlost.

```
//Funkce pro kolizi s nepřítelem
public void kolize_nepritele(Abstraktni_nepratele abstraktniNepratele)
{
    //Kolize s jezdícím krunýřem Klunga
    if(abstraktniNepratele instanceof Klungo &&((Klungo)
abstraktniNepratele).klungoSoucasnyStav == klungoStav.jezdici_krunyr)
    {
        balsacZnicit = true;

        //Spuštění zvukového souboru
        Hra.assetManager.get("audio/zvuky/zabiti.wav", Sound.class).play();

        //Přidání score
        Hud.pridej_score(200);

        //Nastavení bitů, se kterými objekt koliduje na prázdnou hodnotu
        balsacFilter = new Filter();
        balsacFilter.maskBits = Hra.prazdny_bit;
        for(Fixture fixture : nepratelebody.getFixtureList())
            fixture.setFilterData(balsacFilter);

    }
    //Kolize s chodícím Klungem a Ballerem
    else
        zpetna_rychlost(true, false);
}
```

Pro zničení Balsaca slouží funkce *skok_na_hlavu()*, pracující s proměnnou `Bruno` **bruno**. Zničení Bruna je stejné jako ve funkci *kolize_nepritele()*.

```
//Funkce pro zničení nepřítele skokem na hlavu
@Override
public void skok_na_hlavu(Bruno bruno)
{
    balsacZnicit = true;

    //Spuštění zvukového souboru
    Hra.assetManager.get("audio/zvuky/zabiti.wav", Sound.class).play();

    //Přidání score
    Hud.pridej_score(200);

    //Nastavení bitů, se kterými objekt koliduje na prázdnou hodnotu
    balsacFilter = new Filter();
    balsacFilter.maskBits = Hra.prazdny_bit;
    for(Fixture fixture : nepratelebody.getFixtureList())
        fixture.setFilterData(balsacFilter);
}
```

Balsac je jako jediný nepřítel schopen zničit sám sebe a to tím, že se utopí. Pro tento účel je vytvořena funkce *utopeni()* a kód zničení Balsaca je opět stejný jako v předchozích případech.

```
//Funkce pro utopení Balsaca
public void utopeni()
{
    balsacZnicit = true;

    //Spuštění zvukového souboru
    Hra.assetManager.get("audio/zvuky/zabiti.wav", Sound.class).play();

    //Nastavení bitů, se kterými objekt koliduje na prázdnou hodnotu
    balsacFilter = new Filter();
    balsacFilter.maskBits = Hra.prazdny_bit;
    for(Fixture fixture : nepratelBody.getFixtureList())
        fixture.setFilterData(balsacFilter);
}
```

Funkce *draw()* je vytvořena pro vykreslování prvku `Batch` na hrací plochu. Pomocí proměnné `boolean` **`balsacZniceno`** se zjistí, jestli je Balsac živý a vykreslí ho. Funkce, kvůli podmínce **`updateCas`** < 1, vykreslí i na sekundu zničeného Balsaca, aby se stihla promítnout animace jeho zničení.

```
//Funkce pro vykreslování
public void draw(Batch batch)
{
    if(!balsacZniceno || updateCas < 1)
        super.draw(batch);
}
```

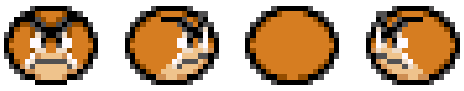
Funkce *update()*, která opět pracuje s časovou proměnnou `float` **`cas`**, pomocí proměnných `boolean` **`balsacZnicit`** a **`balsacZniceno`** zjistí, jestli už je `Body` s nepřítelem zničené. V případě, že není je funkcí *setLinerVelocity()* udělena proměnné `Body` **`nepratelBody`** rychlost, funkcí *setPosition()* je vycetvrována jeho poloha a funkcí *setRegion()* je nastavena animace chůze.

Pokud je proměnná **`balsacZnicit`** nastavena na `true`, je potřeba Balsaca zničit. Zničení Balsaca je zadefinováno zavoláním v proměnné `World` **`world`** funkce *destroyBody()* s parametrem **`nepratelBody`**. Následně musí být proměnná `boolean` **`balsacZniceno`** nastavena na `true` a spuštěna animace mrtvého Balsaca funkcí *setRegion()*.

```
//Funkce pro updatování
public void update(float cas)
{
    updateCas += cas;

    //Zničení nepřítele
    if(balsacZnicit && !balsacZniceno)
    {
        //Zničení Body
        world.destroyBody(nepratelebody);

        //Spuštění zvukového souboru
        setRegion(new
TextureRegion(obrazovka.getTextureAtlas().findRegion("Balsac"), 32, 0, 16,
16));

        updateCas = 0;

        balsacZniceno = true;
    }

    //Update nepřítele, když je živý
    else if(!balsacZniceno)
    {
        //Nastavení rychlosti
        nepratelebody.setLinearVelocity(rychlostNepratele);
        //Nastavení pozice - rozměry podělené dvěma z důvodu vycentrování
        setPosition(nepratelebody.getPosition().x - getWidth() / 2,
nepratelebody.getPosition().y - getHeight() / 2);
        //Nastavení animace
        setRegion(balsacAnimace.getKeyFrame(updateCas, true));
    }
}
```

4.6.2 Ballero

Ballero (Obr.19) je typ nepřítele, který se pohybuje pouze v Y-ové ose a samostatný Bruno ho nedokáže sám zničit.



Obr. 19: Ballero

Třída obsahuje veškeré funkce jako třída Balsac. Funkce jsou téměř identické, proto jsou zde uvedy jen změny v kódu.

V konstruktoru je pro Ballera deklarovaný pohyb pouze v ose Y.

```
//Konstruktor třídy
public Ballero(Obrazovka balleroObrazovka, float x, float y)
{
    //Napojení na herní svět
    super(balleroObrazovka, x + 8 / Hra.korekce, y + 8 / Hra.korekce );

    //Deklarace rychlosti a směru pohybu
    rychlostNepratele = new Vector2(0, 1);
}
```

V kontaktní funkci *kolize_nepritele()* jsou vyobrazeny situace, kdy Ballera zabije Klungo v jezdícím krunýři a situace kdy se Ballero potká s chodícím Klungem nebo jakýmkoliv jiným nepřítelem a začne vykonávat reverzní pohyb.

```
//Funkce pro kolizi s nepřítelem
public void kolize_nepritele(Abstraktni_nepratele abstraktniNepratele)
{
    //Kolize s jezdícím krunýřem Klunga
    if(abstraktniNepratele instanceof Klungo &&((Klungo)
abstraktniNepratele).klungoSoucasnyStav == klungoStav.jezdici_krunyr)
    {
        balleroZnicit = true;

        //Spuštění zvukového souboru
        Hra.assetManager.get("audio/zvuky/zabiti.wav", Sound.class).play();

        //Přidání score
        Hud.pridej_score(300);

        //Nastavení bitů, se kterými objekt koliduje na prázdnou hodnotu
        balleroFilter = new Filter();
        balleroFilter.maskBits = Hra.prazdny_bit;
        for(Fixture fixture : nepratele Body.getFixtureList())
            fixture.setFilterData(balleroFilter);
    }

    //Kolize s chodícím Klungem a Balsacem
    else
        zpetna_rychlost(false, true);
}
```

Jelikož Ballera Bruno nemůže sám zabít je funkce *skok_na_hlavu()* prázdná, ze třídy nejde smazat, z důvodu, že třída je odděna od abstraktní třídy Abstrakni_nepratele.

```
//Funkce pro zničení nepřítele skokem na hlavu
@Override
public void skok_na_hlavu(Bruno bruno)
{
}
```

4.6.3 Klungo

Třetí typ nepřítele, který se ve hře vyskutuje se jmenuje Klungo (Obr.20). Klungo je složitější typ nepřítele než předchozí dva. Skokem na hlavu Klunga, se nepřítel nezničí, ale pouze se změní jeho stav (Klungo se schová do krunýře). Při bočním kontaktu Klunga v krunýři je Klungovi udělen opačný pohyb. Při tomto pohybu Klungo může zabít Bruna, ale i Balsaca, Ballera a i jiného Klunga.

Obr. 20: Klungo

V konstruktoru třídy probíhá napojení na herní svět, deklarace vektoru rychlosti a zadefinování proměnných `klungoStav` **`klungoSoucasnyStav`** a **`klungoPredchoziStav`** na stav: `chuze`.

```
//Konstruktor třídy
public Klungo(Obrazovka klungoObrazovka, float x, float y)
{
    //Napojení na herní svět
    super(klungoObrazovka, x, y);
    //Deklarace rychlosti a směru pohybu
    rychlostNepratele = new Vector2(1, 0);

    //Deklarace výchozích stavů
    klungoSoucasnyStav = klungoPredchoziStav = klungoStav.chuze;
}
```

Klungo stejně jako Bruno mění svoje stavy, se kterými se mění aji jeho animace a proto je i zde obdobně jako ve třídě Bruno vytvořena funkce `TextureRegion` *animace2()*, která rozhoduje, pro jaký stav se mění animace.

```
//Funkce pro animace(jednotlivé stavy)
public TextureRegion animace2(float cas)
{
    TextureRegion textureRegion;

    switch (klungoSoucasnyStav)
    {
        case chuze:
        default:
            textureRegion = klungoAnimace.getKeyFrame(stavovyCas, true);
            break;

        case stojici_krunyr:
        case jezdici_krunyr:
            textureRegion = klungoKrunyr;
            break;
    }

    //Zrcadlové promítnutí animace podle směru pohybu
    if(rychlostNepratele.x > 0 && textureRegion.isFlipX() == false)
    {
        textureRegion.flip(true, false);
    }
    if(rychlostNepratele.x < 0 && textureRegion.isFlipX() == true)
    {
        textureRegion.flip(true, false);
    }

    //Pokud je současný shodný s předchozím --> (stavovyCas + cas) = 0
    stavovyCas = klungoSoucasnyStav == klungoPredchoziStav ? stavovyCas + cas
: 0;

    updateCas =+ cas;

    klungoPredchoziStav = klungoSoucasnyStav;

    return textureRegion;
}
```

Ve třídě *uderNepritele()*, pracující s proměnnou `NeprateleDef` **`neprateleDef`**, se řeší kontakt s nepřáteli. Podmínkou je zjištěno jestli instance třídy `nepprateleDef` je Klungo tzn., jestli nepřítel přicházející do kontaktu je Klungo. V tomto případě je zde ještě vnořená podmínka, která zohledňuje situaci, kdy je Klungo zabit Klungem ve stavu: jezdici_krunyr, tzn. stav nepřátelského Klunga je pohybující se krunýř a současně stav prvního Klunga není pohybující se krunýř. V tomto případě je zavolána funkce *zabiti()*.

V else if větvi je situace, když je Klungo v pohybujícím se krunýři v kontaktu s chodícím Klungem a v takovém stavu je z hlediska kontaktu s ostatními nepřáteli nezničitelný.

V else větvi vnořeného cyklus je situace, kdy se střenou dva Klungové v jezdícím krunýři a začnou vykonávat reverzní pohyb.

V else větvi je zobrazena situace, kdy se potkájí dva chodící Klungové nebo chodící Klungo s Balscem a při tomto kontaktu začnou vykonávat reverzní pohyb, tzn. je zavolána funkce *zpetnaRychlost()* s parametrem true pro x-vou osu.

```
//Funkce pro kolizi s nepřítelem
public void kolize_nepritele(Abstraktni_nepratele abstraktniNepratele)
{
    //Kontakt s Klungem
    if(abstraktniNepratele instanceof Klungo)
    {
        //Kontakt s Klungem: jezdící krunýř
        if( klungoSoucasnyStav != klungoStav.jezdici_krunyr && ((Klungo)
abstraktniNepratele).klungoSoucasnyStav == klungoStav.jezdici_krunyr)
        {
            zabiti();
        }

        //Kontakt s chodícím Klungem
        else
            zpetna_rychlost(true, false);
    }

    //Kontakt s ostatními nepřáteli
    else if(klungoSoucasnyStav != klungoStav.jezdici_krunyr)
        zpetna_rychlost(true, false);
}
```

Funkce *skok_na_hlavu()* porovná aktuální hodnotu proměnné `klungoStav` **`klungoSoucasnyStav`** se stavem stojícího krunýře Klunga. Pokud se Klungo ještě v tomto stavu nenacházel, je do **`klungoSoucasnyStav`** tato hodnota načtena a proměnné **`rychlostNepratele.x`** je nastavena hodnota 0.

V else větvi se řeší situace, kdy už se proměnná nachází ve stavu stojícího krunýře. Z proměnné `Bruno` **`bruno`** se pomocí funkce *getx()* zjistí hodnota polohy Bruna v ose x a porovná se s x-vou hodnotou Klunga. Pokud je Brunova hodnota x menší než u Klunga, znamená to, že Bruno Klunga kopnul směrem doprava a je tedy Klungovi udělena rychlost do pravé strany: `napravo`, v opačném případě rychlost do levé strany.

```
//Funkce pro zničení nepřítele skokem na hlavu
@Override
public void skok_na_hlavu(Bruno bruno)
{
    //klungo se zabalí do krunýře
    if(klungoSoucasnyStav != klungoStav.stojici_krunyr)
    {
        klungoSoucasnyStav = klungoStav.stojici_krunyr;
        rychlostNepratele.x = 0;
    }
    //Detekce z jaké strany je Klungo v krunýři kontaktovaný
    else
    {
        kopnuti(bruno.getX() <= this.getX() ? napravo : nalevo);
    }
}
```

Funkce *kopnuti()*, pracující s proměnnou pomocí proměnné `int` **`rychlostKopu`** přiřadí Klungovi rychlost v X-ové ose a změní její současný stav ze stojícího krunýře na jezdící.

```
//Funkce pro udělení rychlosti Klungovi v krunýři
public void kopnuti(int rychlostKopu)
{
    rychlostNepratele.x = rychlostKopu;
    klungoSoucasnyStav = klungoStav.jezdici_krunyr;
}
```

Funkce ziskej *SoucasnyStav()* vrací pouze hodnotu proměnné `klungoSoucasnyStav`.

```
//Funkce pro získání současného stavu Klunga
public klungoStav ziskejSoucasnyStav()
{
    return klungoSoucasnyStav;
}
```

Funkce *zabiti()*, slouží ke zničení Klunga. Prvně je nastavena proměnná `boolean` **`klungoZnicit`** na `true` a spuštěn zvukový soubor. Jeho Body je také udělen ipuls výskoku a zavolána funkce pro přidání score. V posledním kroku jsou Klungogi obdobně jako u předchozích nepřátel předefinovány jeho `maskBits` na `prazdny_bit`, aby nemohl interagovat s okolím.

```
//Funkce pro zničení Klunga
public void zabiti()
{
    klungoZnicit = true;

    //Spuštění zvukového souboru
    Hra.assetManager.get("audio/zvuky/zabiti.wav", Sound.class).play();

    //Body nepřítele před zničením vyskočí
    nepratelBody.applyLinearImpulse(new Vector2(0, 3f),
nepratelBody.getWorldCenter(), true);

    //Přidání score
    Hud.pridej_score(400);

    //Nastavení bitů, se kterými objekt koliduje na prázdnou hodnotu
    klungoFilter = new Filter();
    klungoFilter.maskBits = Hra.prazdny_bit;
    for(Fixture fixture : nepratelBody.getFixtureList())
        fixture.setFilterData(klungoFilter);
}
```

Třída *update()* samozřejmě musí pracovat s časovou proměnou `float` **cas**, která se používá pro aktualizaci stavové animace, funkce *animace2()* a pro přičítání k další časové proměnné `float` **updateCas**, která je následně použita k odpočtu času, než je Klungo zničen.

Nejprve se ověří, jestli je Klungo v krunýří, odpočítá 5s a poté změní status zpět na chůzi a udělí Klungovi rychlost.

Poté se řeší situace kdy má být Klungo zničen, tzn. proměnná klungoZnicit musí být `true` a současně proměnná **klungoZniceno** musí být `false`. V takovém případě se přičte proměnné `float` **klungoRotace** hodnota 2 a pomocí funkce *rotate()* s parametrem **klungoRotace** Klunga roztočí. Vnořená podmínka pomocí proměnné `float` **updateCas** odpočítá 5s a zároveň pomocí proměnné `boolean` **klungoZniceno** ověří, jestli Body už není zničeno. V tomto případě je zavolána funkce *destroyBody()* s parametrem **nepratelBody** do proměnné `World` **world** a proměnná klungo**Zniceno** je nastavena na true.

V else větvi probíhá jen update rychlosti **nepratelBody** pomocí funkce *setLinearVelocity()* s parametrem **rychlostNepratele**.

```
//Funkce pro updatování
@Override
public void update(float cas)
{
    setRegion(animace2(cas));

    updateCas += cas;

    //Nastavení pozice - rozměry podělené dvěma z důvodu vycentrování
    setPosition(nepratelebody.getPosition().x - getWidth() / 2,
nepratelebody.getPosition().y - 8 / Hra.korekce);

    //Klungo po 5ti vteřinách v krunýři začne znovu chodit
    if(klungoSoucasnyStav == klungoStav.stojici_krunyr && stavovyCas > 5)
    {
        klungoSoucasnyStav = klungoStav.chuze;

        rychlostNepratele.x = 1;

        updateCas = 0;
    }

    //Zničení nepřítele
    if(klungoZnicit && !klungoZniceno)
    {
        //Nastavení rotace
        klungoRotace += 2;
        rotate(klungoRotace);

        //Nepřítel je zničen až po 5ti vteřinách
        if(updateCas > 5 && !klungoZniceno)
        {
            //Zničení Body
            world.destroyBody(nepratelebody);

            klungoZniceno = true;

            updateCas = 0;
        }
    }
    else
    nepratelebody.setLinearVelocity(rychlostNepratele);
}
```

4.6.4 Vor

Poslední třída oděděná od třídy Abstraktni_nepratele se jmenuje Vor (Obr.21) a prakticky se o žádného nepřítele nejedná. Třída ale využívá podobné funkce jako ostatní nepřátele, proto je oděděna od společné třídy.

Obr. 21: Vor

Konstruktor třídy je obdobmý jako u ostatních nepřátel, dochází zde k napojení na herní svět a k deklaraci rychlosti.

```
        //Konstruktor třídy
        public Vor(Obrazovka vorObrazovka, float x, float y)
        {
            //Napojení na herní svět
            super(vorObrazovka, x, y);
            //Deklarace rychlosti a směru pohybu
            rychlostNepratele = new Vector2(1, 0);
        }
```

Aktualizační funkce udatuje polohu, rychlost a animace Body.

```
//Funkce pro updatování
public void update(float cas)
{
    updateCas += cas;

    {
        //Nastavení rychlosti
        nepratelebody.setLinearVelocity(rychlostNepratele);
        //Nastavení pozice - rozměry podělené dvěma z důvodu vycentrování
        setPosition(nepratelebody.getPosition().x - getWidth() / 2,
nepratelebody.getPosition().y - getHeight() / 2);
        //Nastavení animace
        setRegion(vorAnimace.getKeyFrame(updateCas, true));
    }
}
```

Třída obsahuje ještě dvě nevyužité funkce, které zde ale musí být obsaženy, z toho důvodu, že je třída oděděna ze třídy Abstraktni_nepratele.

```
//Funkce pro kolizi s nepřítelem
public void kolize_nepritele(Abstraktni_nepratele abstraktniNepratele)
{
}

//Funkce pro zničení nepřítele skokem na hlavu
@Override
public void skok_na_hlavu(Bruno bruno)
{
}
```

4.7 Třída Abstraktni_mapa

Abstraktní třída Abstraktni_mapa je jedna ze tří tříd tohoto typu a slouží pro definování obdobných vlastností pro dceřiné třídy a pro komunikaci mezi ostatními třídami.

Konstruktor třídy pracuje s proměnnými `Obrazovka` **`mapaObrazovka`** a `MapOject` **`MapObject`**. Prvně se zde nachází propojení s herním světěm, pomocí prvku this a zavolána funkce pro definování vlastností, která je ovšem deklarována až v dceřiných třídách.

```
//Konstruktor třídy
public Abstraktni_mapa(Obrazovka mapaObrazovka, MapObject mapObject)
{
    //Napojení na herní svět
    this.mapObject = mapObject;
    this.obrazovka = mapaObrazovka;
    this.world = mapaObrazovka.getWorld();
    this.tiledMap = mapaObrazovka.getTiledMap();
    this.rectangle = ((RectangleMapObject) mapObject).getRectangle();

    definice_mapa();
}
```

Funkce *ziskani_id()* je vytvořena, z důvodu, že po úderu hlavou velkého Bruna do cihly, je potřeba odstranit grafickou texturu cihly. Právě tato funkce slouží ke zjištění, která cihla byla odstraněna. Do proměnné `TiledMapTileLayer` **`tiledMapTileLayer`** je alokována vrstva 1 z editoru Tiled, který zastupuje veškeré grafické textury objektů.

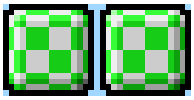
```
//Funkce pro získání id mince nebo cihle, kterou Bruno udeřil hlavou
public TiledMapTileLayer.Cell ziskani_id()
{
    //Alokování vrstvy, do které jsou nakreslené mince resp. cihle
    tiledMapTileLayer = (TiledMapTileLayer) tiledMap.getLayers().get(1);

    //Vrácení id cihle + pixelová korekce
    return
tiledMapTileLayer.getCell((int)(mapaBody.getPosition().x*Hra.korekce / 16),
(int)(mapaBody.getPosition().y*Hra.korekce / 16));
}
```

Jsou zde ještě deklarovány dvě funkce pro dceřiné třídy: definice_mapa() a uder_hlava().

```
//Funkce pro definování vlastností
public abstract void definice_mapa();

//Funkce pro kontakt hlavy Bruna s mincí nebo cihlou
public abstract void uder_hlava(Bruno bruno);
```

4.7.1 Cihla

Cihla (Obr.22) je mapový oběkt po, kterém je Bruno schopen skákat, když je Bruno zvětšený, může cihlu za 50 bodů zničit.



Obr. 22: Cihla

Konstruktor třídy pracuje s proměnnými `Obrazovka` **`CihlaObrazovka`** a `MapObject` **`mapObject`**, pomocí kterých se napojí na herní svět.

```
        //Konstruktor třídy
        public Cihla(Obrazovka cihlaObrazovka, MapObject mapObject)
        {
            //Napojení na herní svět
            super(cihlaObrazovka, mapObject);

            fixture.setUserData(this);
        }
```

Funkce *definice_mapa()* slouží pro definování vlastností a je obdobná stejné funkci ve tříde Bruno, s rozdílem že `BodyType` je deklarovaný na `StaticBody`, jelikož se cihla nehýbe.

```
//Funkce pro definování vlastností
@Override
public void definice_mapa()
{
    cihlaDefiniceBody = new BodyDef();
    cihlaDefiniceFixture = new FixtureDef();
    cihlaTelo = new PolygonShape();

    //Definice typu Body
    cihlaDefiniceBody.type = BodyDef.BodyType.StaticBody;
    //Definice polohy
    cihlaDefiniceBody.position.set((rectangle.getX() + rectangle.getWidth() /
2) / Hra.korekce, (rectangle.getY() + rectangle.getHeight() / 2) /
Hra.korekce);
    //Vytvoření Body
    mapaBody = world.createBody(cihlaDefiniceBody);

    //Vytvoření čtvercového Body
    cihlaTelo.setAsBox(rectangle.getWidth() / 2 / Hra.korekce,
rectangle.getHeight() / 2 / Hra.korekce);
    cihlaDefiniceFixture.shape = cihlaTelo;

    //Definice bitů, které objekt zastupují
    cihlaDefiniceFixture.filter.categoryBits = Hra.cihla_bit;

    //Definice bitů, se kterými je objekt v kontaktu
    cihlaDefiniceFixture.filter.maskBits = Hra.bruno_bit | Hra.objekty_bit |
Hra.zeme_bit | Hra.mince_bit | Hra.cihla_bit | Hra.ballero_bit |
Hra.hlava_bruna_bit;

    //Vytvoření Fixture
    fixture = mapaBody.createFixture(cihlaDefiniceFixture);
}
```

Funkce *uder_hlava()* je zavolána v případě, že je Bruno velký a hlavou kontaktuje cihlu. Prvně je zavolána funkce ze třídy Abstraknti_mapa: *ziskani_id()*, která zjistí, jaká cihla byla kontaktována a pomocí funkce *setTile()* s parametrem `null` ji vymaže. Obdobně jako v předchozích případech je zničení doprovázeno zvukovým souborem a ze třídy HUD je zavolána funkce *pridej_score()*, která zvýší score o 50.

Cihly jsou pak nastaveny `categoryBits` na `zniceny_bit`, která zabrání zničené cihle další interakci s herním světem. V else větvi je zohledněna situaci kdy je bruno malý a kontaktuje cihlu, v takovém případě je pouze spuštěn zvukový soubor.

```
//Funkce pro interakci vody se třídou Bruno
@Override
public void uder_hlava(Bruno bruno)
{
    //Cihlu může zničit pouze velký Bruno
    if(bruno.jeVelky())
    {
        ziskani_id().setTile(null);

        //Spuštění zvukového souboru
        Hra.assetManager.get("audio/zvuky/znicenacihla.wav",
Sound.class).play();

        //Přidání score
        Hud.pridej_score(50);

        //Předefinování bitů na nulové
        cihlaFilter = new Filter();
        cihlaFilter.categoryBits = Hra.zniceny_bit;
        fixture.setFilterData(cihlaFilter);
    }

    else
    //Spuštění zvukového souboru
    Hra.assetManager.get("audio/zvuky/neznicenacihla.wav",
Sound.class).play();
}
```

4.7.2 Mince

Mince (Obr.23) je součást mapy, označená zeleným nápisem ON, po tom ho Bruno kontaktuje svojí hlavou, změní se jeho grafická textura na červený nápis OFF. Po kontaktu mohou nastat tři situace. Základní situace je, že se zvýší score hráče o 100, nebo se zobrazí dva typy bonusových předmětů: voda, bonus. Předměty následně začnou vykonávat pohyb směrem nahoru. Třída obsahuje obdobné funkce jako třída Cihla, proto jsou zde uvedeny jen funkce s odlišným kódem.

Obr. 23: Mince

V konstruktoru třídy dochází obdobně jako ve třídě Cihla, pomocí proměnných `Obrazovka` **`minceObrazovka`** a `MapObject` **`object`**, na propojení s herním světem. Když Bruno hlavou kontaktuje blok s mincí, změní se grafika mince. Pro tento účel je vytvořena proměnná `TiledMapTileSet` **`tilesMapTileSet`**, ke které je alokováný soubor s grafickými texturami, vytvořenými v programu Tiled, obsahující právě tu texturu, která se má po kontaktu zobrazit.

```
//Konstruktor třídy
public Mince(Obrazovka minceObrazovka, MapObject object)
{
    //Napojení na herní svět
    super(minceObrazovka, object);

    //Animace prazdne mince
    tiledMapTileSet = tiledMap.getTileSets().getTileSet("bloky");
}
```

Funkce uder_hlava() je naimplementovaná ze třídy abstrakni_mapa a pracuje s proměnou `Bruno` **`bruno`**. Třída vyobrazuje čtyři eventuální situace, když Bruno hlavou udeří blok s mincí, blok s bonusovou položkou: voda, blok s bonusovou položkou: bonus nebo blok s již sebranou mincí. Jsou zde zavolány funkce *ziskaniId()*, *getTile()* a *getId()*, které jsou porovnány s proměnnou `int` **`prazdna_mince`**, která deklaruje již vybranou minci. V else větvi se nachází vnořená podmínka, kde jsou do proměnné `MapObject` **`object`**, zavolány funkce *getProperties()* a *constainsKey()* s parametrem voda. Parametr voda je deklarován v programu Tiled, pomocí kolonky custom propertiers a zjišťuje, jestli se v políčku nenachází bonusová položka. Pokud ano je do proměnné `Obrazovka` **`obrazovka`** zavolána funkce *vytvorVec()*, pomocí které se zobrazí položka. Zobrazení je opět doprovázeno zvukem. Naprosto obdobně je udělána druhá větev pro druhou bonusovou položku: bonus.

V else podmínce je zobrazena situace, kdy Bruno najde políčko s mincí. Pomocí funkcí *getCell()*, *setTile()* se nastaví změna grafické textury. A do proměnné Hud se zavolá funkce *pridejScore()* s parametrem 100, která zvýší hráčovo score o 100.

```
//Funkce pro interakci vody se třídou Bruno
@Override
public void uder_hlava(Bruno bruno)
{
    //Kontakt s již prázdnou mincí
    if(ziskani_id().getTile().getId() == prazdna_mince)
        Hra.assetManager.get("audio/zvuky/neznicenacihla.wav",
Sound.class).play();

    else
    {
        //Nastavení grafiky prázdné mince
        ziskani_id().setTile(tiledMapTileSet.getTile(prazdna_mince));

        //Kontakt s položkou voda
        if(mapObject.getProperties().containsKey("voda"))
        {
            //Vytvoření vody
            obrazovka.vytvor_voda(new Voda_zobrazovani(new
Vector2(mapaBody.getPosition().x, mapaBody.getPosition().y + 16 /
Hra.korekce), Voda.class));
            //Spuštění zvukového souboru
            Hra.assetManager.get("audio/zvuky/predmet.wav",
Sound.class).play();
        }
        //Kontakt s položkou bonus
        else if(mapObject.getProperties().containsKey("bonus"))
        {
            //Vytvoření bonusu
            obrazovka.vytvor_bonus(new Bonus_zobrazovani(new
Vector2(mapaBody.getPosition().x, mapaBody.getPosition().y + 16 /
Hra.korekce), Bonus.class));
            //Spuštění zvukového souboru
            Hra.assetManager.get("audio/zvuky/predmet.wav",
Sound.class).play();
        }
        else
        {
            //Spuštění zvukového souboru
            Hra.assetManager.get("audio/zvuky/mince.wav", Sound.class).play();
           //Přidání score
            Hud.pridej_score(100);
        }
    }
}
```

4.8 Třída Abstraktni_polozky

Poslední abstraktní třída se jmenuje Abstraktni_polozky a slouží pro oddědění tříd, které definují bonusové položky schopné interakce s Brunem.

Třída je rozšířena o prvek `Sprite` z pluginu libGDX.

```
public abstract class Abstraktni_polozky extends Sprite
```

Konstruktor třídy obsahuje parametr `Obrazovka` **`polozkyObrazovka`**, pomocí kterého ja napojen na herní svět. Jsou zde také dva polohové parametry `float` **`x`**, `float` **`y`**, které jsou použity ve funkci *setPosition()* a definijů polohu položky.

Funkce *definice_vec()* bude volána v dceřiných třídách a slouží k nastavení vlastnostností jednotlivých položek.

```
//Konstruktor třídy
public Abstraktni_polozky(Obrazovka polozkyObrazovka, float x, float y)
{
    //Napojení na herní svět
    this.obrazovka = polozkyObrazovka;
    this.world = polozkyObrazovka.getWorld();

    //Funkce deklarující polohu
    setPosition(x, y);

    definice_vec();
}
```

Abstraktní třída obsahuje ještě ctyři funkce, které budou následně odděděny do dceřiných tříd.

```
//Funkce pro definování vlastností
public abstract void definice_vec();

//Funkce pro interakci s Brunem
public abstract void pouzit_vodu(Bruno bruno);

//Funkce pro interakci s Brunem
public abstract void pouzit_bonus(Bruno bruno);

//Funkce pro updatování
public abstract void update(float cas);
```

4.8.1 Voda a Bonus

Třídy obou položek jsou téměř identické, proto zde bude vysvětlena pouze třída Voda. Obě třídy jsou odděděny z abstraktní třídy: Abstraktni_mapa. Grafická textura vody je zobrazena na obr.24 a bonusu na obr.25.

Obr. 24: Voda

Obr. 25: Bonus

V konstruktoru třídy dochází k napojení na herní svět, deklarování rychlosti objektu směrem nohoru a zavolání funkce *animaceVoda()*.

```
        //Konstruktor třídy
        public Voda(Obrazovka vodaObrazovka, float x, float y)
        {
            //Napojení na herní svět
            super(vodaObrazovka, x, y);

            //Deklarace rychlosti a směru pohybu
            rychlostVody = new Vector2(0,0.3f);

            //Funkce pro animace
            animaceVoda();
        }
```

Funkce pro animace je vytvořena opět obdobně jako v ostatních třídách.

```
//Funkce pro animace
public void animaceVoda()
{
    //Alokování souboru s animací
    setRegion(obrazovka.getTextureAtlas().findRegion("Voda"), 0, 0, 16, 13);
    //Definování reálných okrajů textur
    setBounds(getX(), getY(), 16 / Hra.korekce, 16 / Hra.korekce);
}
```

Funkce pro definování vlastností: *definice_vec()*.

```
//Funkce pro definování vlastností
@Override
public void definice_vec()
{
    vodaDefiniceBody = new BodyDef();
    vodaDefiniceFixtur = new FixtureDef();
    vodaTelo = new CircleShape();

    //Definice typu Body
    vodaDefiniceBody.type = BodyDef.BodyType.DynamicBody;
    //Definice polohy
    vodaDefiniceBody.position.set(getX(), getY());
    //Vytvoření Body
    vodaBody = world.createBody(vodaDefiniceBody);

    //Vytvoření kruhového Body
    vodaTelo.setRadius(6 / Hra.korekce);
    vodaDefiniceFixtur.shape = vodaTelo;

    //Definice bitů, které objekt zastupují
    vodaDefiniceFixtur.filter.categoryBits = Hra.vec_bit;

    //Definice bitů, se kterými je objekt v kontaktu
    vodaDefiniceFixtur.filter.maskBits = Hra.bruno_bit | Hra.objekty_bit |
Hra.zeme_bit | Hra.mince_bit | Hra.cihla_bit;

    //Vytvoření Fixture
    vodaBody.createFixture(vodaDefiniceFixtur).setUserData(this);
}
```

Funkce pro zničení položky deklaruje proměnné `boolean` **`vodaZnicit`** hodnotu `true`.

```
//Funkce pro zničení položky
public void zniceni()
{
    vodaZnicit = true;
}
```

Funkce pro interakci v Brunem: *pouzit_vodu()* zavolá funkci *zniceni()* a do proměnné Bruno bruno funkci *rust()*, která zvětší velikost Bruna na dvojnásobek.

```
//Funkce pro interakci vody se třídou Bruno
@Override
public void pouzit_vodu(Bruno bruno)
{
    zniceni();
    bruno.rust();
}
```

Funkce pro vykreslování: *draw()*, funguje jen, když položka není zničena.

```
//Funkce pro vykreslování
public void draw(Batch batch)
{
    if(!vodaZniceno)
        super.draw(batch);
}
```

Aktualizační funkce, odbodně jako u nepřátel zohledňuje situaci, kdy Bruno položku sebral a objekt musí být zničet. V else větvi je aktualizována poloha, rychlost a animace položky.

```
//Funkce pro updatování
public void update(float cas)
{
    //Zničení položky
    if(vodaZnicit && !vodaZniceno)
    {
        //Zničení Body
        world.destroyBody(vodaBody);

        vodaZniceno = true;
    }

    //Update položky, když je živá
    else if(!vodaZniceno)
    {
        //Nastavení rychlosti
        vodaBody.setLinearVelocity(rychlostVody);
        //Nastavení pozice - rozměry podělené dvěma z důvodu vycentrování
        setPosition(vodaBody.getPosition().x - getWidth() / 2,
vodaBody.getPosition().y - getHeight() / 2);
    }
}
```

Jelikož se jedná opět o odděděnou třídu, musí zde být z důvodu kompilace i funkce, která funguje jenom pro druhou dceřinou třídu: Bonus.

```
//Funkce pro interakci bonusu se třídou Bruno
@Override
public void pouzit_bonus(Bruno bruno)
{
}
```

4.9 Třída Voda_zbrazovani a Bonus_zobrazování

Tyto třídy jsou vytvořené pro zobrazení jednotlivých položek v herním světě. Třídy jsou opět identické, proto zde bude vysvětlena pouze třída Voda_zobrazovani.

Třída obsahuje pouze konstruktor, který je volán ze třídy Obrazovka a funguje tak, že na požadovanou pozici vytvoří požadovanou třídu.

```
//Konstruktor třídy
public Voda_zobrazovani(Vector2 pozice, Class<?> typ)
{
    //Funkce zobrazí na pozici třídu
    this.pozice = pozice;
    this.typ = typ;
}
```

4.10 Třída Vytvoreni_sveta

Java třída Vytvoreni_sveta zajišťuje propojení grafických textur, vytvořených v programu Tiled, s kontaktní logikou v Android Studiu.

Pro každou vrstvu `Object Layer`, z programu Tiled, je vytvořena for podmínka, která pro všechny mapové objekty: `MapObject` **`mapObject`** z proměnných `TiledMap` **`tiledMap`**, funkcemi *getLayers()*, *getObjects()* a *get()*, s indexem vrstvy z programu Tiled, alokoje patřičné vrstvy. Funkce *getByType()* s parametrem **`RectangleMapObject.class`** deklaruje vrstvu jako obdelníkovou. Následně je vytvořena proměnná `Rectangle` **`rectangle`**, do které je funkcí *getRectangle()*, zavolanou do proměnné **`mapObject`**, alokována obdelníková struktůra vrstvy.

Ke každému prvku `Body` **`body`**, jsou přiřazeny jeho vlastnosti pomocí prvků `Bodydef` **`bodyDef`** a `FixtureDef` **`fixtureDef`**, které jsou následně do **`body`** vloženy funkcemi *createBody()* a *createfixture()*.

```
//Konstruktor třídy
public Vytvoreni_sveta(Obrazovka obrazovka)
{
    //Napojení na herní svět
    world = obrazovka.getWorld();
    tiledMap = obrazovka.getTiledMap();

    bodyDef = new BodyDef();
    polygonShape = new PolygonShape();
    fixtureDef = new FixtureDef();

    //Balsac
    balsac = new Array<Balsac>();

    for(MapObject mapObject :
tiledMap.getLayers().get(6).getObjects().getByType(RectangleMapObject.class))
    {
        //Definovány čvetcových mapových struktur
        rectangle  = ((RectangleMapObject) mapObject).getRectangle();

        //Alokace třídy
        balsac.add(new Balsac(obrazovka, rectangle.getX() / Hra.korekce,
rectangle.getY() / Hra.korekce));

    }

    //klungo
    klungo = new Array<Klungo>();

    for(MapObject mapObject :
tiledMap.getLayers().get(7).getObjects().getByType(RectangleMapObject.class))
    {
        //Definovány čvetcových mapových struktur
        rectangle  = ((RectangleMapObject) mapObject).getRectangle();

        //Alokace třídy
        klungo.add(new Klungo(obrazovka, rectangle.getX() / Hra.korekce,
rectangle.getY() / Hra.korekce));
    }

    //ballero
    ballero = new Array<Ballero>();

    for(MapObject mapObject :
tiledMap.getLayers().get(8).getObjects().getByType(RectangleMapObject.class))
    {
        //Definovány čvetcových mapových struktur
        rectangle  = ((RectangleMapObject) mapObject).getRectangle();

        //Alokace třídy
        ballero.add(new Ballero(obrazovka, rectangle.getX() / Hra.korekce,
rectangle.getY() / Hra.korekce));

    }
```

```
    //vor
    vor = new Array<Vor>();

    for(MapObject mapObject :
tiledMap.getLayers().get(10).getObjects().getByType(RectangleMapObject.class))
    {
        //Definovány čvetcových mapových struktur
        rectangle  = ((RectangleMapObject) mapObject).getRectangle();

        //Alokace třídy
        vor.add(new Vor(obrazovka, rectangle.getX() / Hra.korekce,
rectangle.getY() / Hra.korekce));

    }

    //Cihla
    for(MapObject mapObject :
tiledMap.getLayers().get(5).getObjects().getByType(RectangleMapObject.class))
    {
        //Alokace třídy
        new Cihla(obrazovka, mapObject);
    }

    //Mince
    for(MapObject mapObject :
tiledMap.getLayers().get(4).getObjects().getByType(RectangleMapObject.class))
    {
        //Alokace třídy
        new Mince(obrazovka, mapObject);
    }

    //Okraje
    for(MapObject mapObject :
tiledMap.getLayers().get(9).getObjects().getByType(RectangleMapObject.class))
    {
        //Definovány čvetcových mapových struktur
        rectangle  = ((RectangleMapObject) mapObject).getRectangle();

        //Definice typu Body
        bodyDef.type = BodyDef.BodyType.StaticBody;
        //Definice polohy
        bodyDef.position.set((rectangle.getX() + rectangle.getWidth() / 2) /
Hra.korekce, (rectangle.getY() + rectangle.getHeight() / 2) / Hra.korekce);
        //Vytvoření Body
        body = world.createBody(bodyDef);

        //Vytvoření čtvercového Body
        polygonShape.setAsBox(rectangle.getWidth() / 2 /
Hra.korekce,rectangle.getHeight() / 2 / Hra.korekce);
        fixtureDef.shape = polygonShape;

        //Definice bitů, které objekt zastupují
        fixtureDef.filter.categoryBits = Hra.okraje_bit;
        //Definice bitů, se kterými je objekt v kontaktu
        fixtureDef.filter.maskBits = Hra.bruno_bit | Hra.vor_hlava_bit |
Hra.nepratele_bit;

        //Vytvoření Fixture
        body.createFixture(fixtureDef);
    }
```

```
    //Země
    for(MapObject mapObject :
tiledMap.getLayers().get(2).getObjects().getByType(RectangleMapObject.class))
    {
        //Definovány čvetcových mapových struktur
        rectangle  = ((RectangleMapObject) mapObject).getRectangle();

        //Definice typu Body
        bodyDef.type = BodyDef.BodyType.StaticBody;
        //Definice polohy
        bodyDef.position.set((rectangle.getX() + rectangle.getWidth() / 2) /
Hra.korekce, (rectangle.getY() + rectangle.getHeight() / 2) / Hra.korekce);
        //Vytvoření Body
        body = world.createBody(bodyDef);

        //Vytvoření čtvercového Body
        polygonShape.setAsBox(rectangle.getWidth() / 2 /
Hra.korekce,rectangle.getHeight() / 2 / Hra.korekce);
        fixtureDef.shape = polygonShape;

        //Definice bitů, které objekt zastupují
        fixtureDef.filter.categoryBits = Hra.zeme_bit;
        //Definice bitů, se kterými je objekt v kontaktu
        fixtureDef.filter.maskBits = Hra.bruno_bit | Hra.vor_hlava_bit |
Hra.nepratele_bit | Hra.ballero_bit | Hra.vor_bit;

        //Vytvoření Fixture
        body.createFixture(fixtureDef);
    }

    //Objekty
    for(MapObject mapObject :
tiledMap.getLayers().get(3).getObjects().getByType(RectangleMapObject.class))
    {
        //Definovány čvetcových mapových struktur
        rectangle  = ((RectangleMapObject) mapObject).getRectangle();

        //Definice typu Body
        bodyDef.type = BodyDef.BodyType.StaticBody;
        //Definice polohy
        bodyDef.position.set((rectangle.getX() + rectangle.getWidth() / 2) /
Hra.korekce, (rectangle.getY() + rectangle.getHeight() / 2) / Hra.korekce);
        //Vytvoření Body
        body = world.createBody(bodyDef);

        //Vytvoření čtvercového Body
        polygonShape.setAsBox(rectangle.getWidth() / 2 /
Hra.korekce,rectangle.getHeight() / 2 / Hra.korekce);
        fixtureDef.shape = polygonShape;

        //Definice bitů, které objekt zastupují
        fixtureDef.filter.categoryBits = Hra.objekty_bit;

        //Vytvoření Fixture
        body.createFixture(fixtureDef);
    }
}
```

Funkce `array<Abstraktni_nepratele>` *zobrazeni_nepratel()* zobrazuje všechny typy nepřátel do herního světa, tím že se do proměnné `Array<Abstraktni_nepratele>` **`nepratele`** se zavolá funkce *add()* s parametrem typu nepřítele.

```
//Funkce pro vytvoření nepřátel
public Array<Abstraktni_nepratele> zobrazeni_nepratel()
{
    nepratele = new Array<Abstraktni_nepratele>();

    nepratele.addAll(balsac);
    nepratele.addAll(klungo);
    nepratele.addAll(ballero);
    nepratele.addAll(vor);

    return nepratele;
}
```

4.11 Třída Kontakty

V této třídě je nastavena veškerá kontaktní logika mezi jednotlivými objekty.

Do třídy je naimplementovaný prvek `ContactListener` z pluginu badlogic.gdx, pomocí kterého jsou do třídy naimplementovány potřebné metody.

```
public class Kontakty implements ContactListener
```

První naimplementovaná třída se nazývá *beginContact()* a slouží k definici, toho, co se stane, když začíná kontakt mezi jednotlivými objekty. Definice kontaktu je prakticky kolize mezi dvěma prvky `Fixture`.

Kontakt probíhá mezi dvěma prvky typu `Fixture`: **`fixA`** a **`FixB`**, do kterých jsou funkcí *getFixtureA()* resp. *getFixtureB()* načteny objekty, které mezi sebou interagují. Do proměnné `int` **`kolize`** jsou načteny hodnoty `categoryBits` pro proměnné **`fixA`** i **`fixB`**. Následně se cyklem switch pro proměnnu **`kolize`** definovaly všechny eventuální situace interakce všech objektů.

Objekty při kontaktu zastupují jejich bity, které jsou uváděny za operátorem case. Následně se pomocí funkce *getFilterData()* zjistí `categoryBits` proměnné **`fixA`**, porovnají se s bity, co mají přicházet do kontaktu a spustí se příslušná funkce. Jelikož půjde vždy o interakci mezi dvěma objekty, musí se v podmínce else pokaždé zohlednit i zrcadlová situace.

```
//funkce pro začátek kontaktu
@Override
public void beginContact(Contact contact)
{
    //kontakt = kolize dvou fixtur
    Fixture fixA = contact.getFixtureA();
    Fixture fixB = contact.getFixtureB();

    //od dvou kolidujících objektů vemu jejich bity, abych je identifikoval
    int kolize = fixA.getFilterData().categoryBits |
fixB.getFilterData().categoryBits;

    switch (kolize)
    {
        //kontakt hlavy Bruna s políčkem mince a cihly
        case Hra.hlava_bruna_bit | Hra.cihla_bit:
        case Hra.hlava_bruna_bit | Hra.mince_bit:
            if(fixA.getFilterData().categoryBits == Hra.hlava_bruna_bit)
                ((Abstraktni_mapa) fixB.getUserData()).uder_hlava((Bruno)
fixA.getUserData());
            else
                ((Abstraktni_mapa) fixA.getUserData()).uder_hlava((Bruno)
fixB.getUserData());
            break;

        //kontakt hlavy nepřítele s Brunem
        case Hra.hlava_nepritele_bit | Hra.bruno_bit:
            if(fixA.getFilterData().categoryBits == Hra.hlava_nepritele_bit)

((Abstraktni_nepratele)fixA.getUserData()).skok_na_hlavu((Bruno)
fixB.getUserData());
            else

((Abstraktni_nepratele)fixB.getUserData()).skok_na_hlavu((Bruno)
fixA.getUserData());
            break;

        //kontakt nepřítele s objektrem
        case Hra.nepratele_bit | Hra.objekty_bit:
            if(fixA.getFilterData().categoryBits == Hra.nepratele_bit)

((Abstraktni_nepratele)fixA.getUserData()).zpetna_rychlost(true, false);
            else

((Abstraktni_nepratele)fixB.getUserData()).zpetna_rychlost(true, false);
            break;

        //kontakt nepřítele se zemí
        case Hra.nepratele_bit | Hra.zeme_bit:
            if(fixA.getFilterData().categoryBits == Hra.nepratele_bit)

((Abstraktni_nepratele)fixA.getUserData()).zpetna_rychlost(true, false);
            else

((Abstraktni_nepratele)fixB.getUserData()).zpetna_rychlost(true, false);
            break;
```

```
        //kontakt Bruna s nepřítelem
        case Hra.bruno_bit | Hra.nepratele_bit:
            if(fixA.getFilterData().categoryBits == Hra.bruno_bit)
                ((Bruno)
fixA.getUserData()).uder((Abstraktni_nepratele)fixB.getUserData());
            else
                ((Bruno)
fixB.getUserData()).uder((Abstraktni_nepratele)fixA.getUserData());
            break;

        //kontakt nepřátel vzájemně
        case Hra.nepratele_bit | Hra.nepratele_bit:

((Abstraktni_nepratele)fixA.getUserData()).kolize_nepritele((Abstraktni_neprat
ele)fixB.getUserData());

((Abstraktni_nepratele)fixB.getUserData()).kolize_nepritele((Abstraktni_neprat
ele)fixA.getUserData());
            break;

        //kontakt věcí a Bruna
        case Hra.vec_bit | Hra.bruno_bit:
            if(fixA.getFilterData().categoryBits == Hra.vec_bit)
                ((Abstraktni_polozky)fixA.getUserData()).pouzit_vodu((Bruno)
fixB.getUserData());
            else
                ((Abstraktni_polozky)fixB.getUserData()).pouzit_vodu((Bruno)
fixA.getUserData());
            break;

        //kontakt věcí a Bruna
        case Hra.bonus_bit | Hra.bruno_bit:
            if(fixA.getFilterData().categoryBits == Hra.bonus_bit)
                ((Abstraktni_polozky)fixA.getUserData()).pouzit_bonus((Bruno)
fixB.getUserData());
            else
                ((Abstraktni_polozky)fixB.getUserData()).pouzit_bonus((Bruno)
fixA.getUserData());
            break;

        //kontakt Ballero x cihle
        case Hra.ballero_bit | Hra.cihla_bit:
            if(fixA.getFilterData().categoryBits == Hra.ballero_bit)

((Abstraktni_nepratele)fixA.getUserData()).zpetna_rychlost(false, true);
            else

((Abstraktni_nepratele)fixB.getUserData()).zpetna_rychlost(false, true);
            break;

        //kontakt Ballero x mince
        case Hra.ballero_bit | Hra.mince_bit:
            if(fixA.getFilterData().categoryBits == Hra.ballero_bit)

((Abstraktni_nepratele)fixA.getUserData()).zpetna_rychlost(false, true);
            else

((Abstraktni_nepratele)fixB.getUserData()).zpetna_rychlost(false, true);
            break;
```

```
        //kontakt Ballero x mince
        case Hra.ballero_bit | Hra.zeme_bit:
            if(fixA.getFilterData().categoryBits == Hra.ballero_bit)

((Abstraktni_nepratele)fixA.getUserData()).zpetna_rychlost(false, true);
            else

((Abstraktni_nepratele)fixB.getUserData()).zpetna_rychlost(false, true);
            break;

        //kontakt Bruno x Vodou
        case Hra.bruno_bit | Hra.okraje_bit:
            if(fixA.getFilterData().categoryBits == Hra.bruno_bit)
                ((Bruno)
fixA.getUserData()).uder((Abstraktni_nepratele)fixB.getUserData());
            else
                ((Bruno)
fixB.getUserData()).uder((Abstraktni_nepratele)fixA.getUserData());
            break;

        //kontakt Vor x Země
        case Hra.vor_hlava_bit | Hra.zeme_bit:
            if(fixA.getFilterData().categoryBits == Hra.vor_hlava_bit)

((Abstraktni_nepratele)fixA.getUserData()).zpetna_rychlost(true, false);
            else

((Abstraktni_nepratele)fixB.getUserData()).zpetna_rychlost(true, false);
            break;


        //kontakt Balsac x Voda
        case Hra.nepratele_bit | Hra.okraje_bit:
            if(fixA.getFilterData().categoryBits == Hra.nepratele_bit)
                ((Balsac)fixA.getUserData()).utopeni();
            else
                ((Balsac)fixB.getUserData()).utopeni();
            break;
    }
```

5 ZÁVĚR

Aplikace, která je v této práci popsána, úspěšně funguje. Všechny její funkce fungují zcela bez problémů. Doplňkový text podrobně uvádí do problematiky vývoje mobilních her či aplikací, a proto práce může prakticky sloužit jako manuál pro vývoj obdobných softwarových aplikací a jako pomůcká ve výuce programování. Zdrojový kód je z důvodu přehlednosti zařazen přímo do textu, i když to prodlužuje celkovou délku textu.

V práci je uveden podrobný princip vývoje mnoha herních prvků, pomocí kterého je možné hru ještě mnohonásobně rozšířit. Příkladem můžou být například různé modifikace nepřátel, ovládání a mapových prvků.

Jako další eventuální rozšíření aplikace přichází v úvahu sdílená databáze se score mezi uživateli. Databáze by mohla řadit uživatele i dle regionu, ve kterém hru hrají a vytvářet lokální řebříčky umístění.

Hra by mohla být rozšířena i o možnost hry pro více uživatelů, tzv. multiplayer. Existovala by tedy další hratelná postava podobná Brunovi, se kterou by druhý uživatel mohl hrát level současně a oba uživatelé by tak mezi sebou mohli soupeřit o vyšší score.

6 SEZNAM POUŽITÉ LITERATURY

[1] Lacko, Luboš. *Vývoj aplikací pro Android.* Praha: Computer Press, 2015. ISBN 978-80-251-4347-6.

[2] Marvan, Filip. *Mobilní operační systém Android*, [online].. [cit.27.7.2011]. Dostupné z: https://diit.cz/clanek/mobilni-operacni-system-android

[3] DiMarzio, Jerome. *Programujeme hry pro android 4.* Praha: Computer Press, 2012. ISBN 978-80-251-3754-3.

[4] Sedláček, Vojtěch. *Programujeme s libGDX*, [online].. [cit.14.7.2017]. Dostupné z: http://tvorbaher.cz/programujeme-s-libgdx/

[5] Frank, jiří. *Lekce 2 – Android Programování – Vývojové prostředí*, [online]... Dostupné z: https://www.itnetwork.cz/java/android/tutorial-programovani-pro-android-v-jave-vyvojove-prostredi

[6] Olupot, Nathan Ernes. *Guide: How To Create Your First Android App With Android Studio*, [online].. [cit.28.5.2016]. Dostupné z: https://pctechmag.com/2016/05/guide-how-to-create-your-first-android-app-with-android-studio/

PŘÍLOHY

Součástí diplomové práce je CD obsahující program aplikace.